Think in Patterns

The Python Coding Interview Notebook

340 Problems · 36 Patterns · From First Principles to AI-Era Rounds

1. Arrays, Strings & Hashing
2. Two Pointers & Sliding Window
3. Searching & Sorting
4. Linked Lists
5. Stacks & Queues
6. Trees
7. Heaps
8. Graphs
9. Recursion & Backtracking
10. Dynamic Programming
11. Greedy & Intervals
12. Math & Bits
13. Design
14. Applied & AI-Era Coding

ProServIQ

Name	: Ravi Teja K
Subject	: DSA Patterns in Python
Pages	: 2288

Contents

Start here

Copyright and licence 5
Before you start 7
How to use this bundle 9
Three ways through this book 13
Route A: Foundations first 15
Route B: Interview core 17
Route C: Applied (AI-era) 20

Foundations

What DSA Is, and Why Interviews Ask It 22
Big-O Without Fear 28
Memory, Variables and References 35
Arrays and Python Lists 41
Strings: Text as a Row of Characters 47
Hash Maps: Finding Things Instantly 53
Linked Lists: a Treasure Hunt of Pointers 59
Stacks and Queues 65
Recursion: Solving Smaller Copies 70
Sorting and Searching: Order Makes Things Fast 76
Trees: Things Inside Things 82
Heaps and Priority Queues 88
Graphs: Maps of Connections 95
Answers: Foundations 101

Interview toolkit

The interview framework 104
Which pattern is this? 107
Big-O and Python at a glance 111
The 8-week plan 114
Online Assessments: What to Expect 119
Interviews in the AI era 122

ProServIQ

Fun fact Edsger Dijkstra designed his shortest-path algorithm in about twenty minutes, at a café in Amsterdam in 1956.

Arrays, Strings & Hashing

1. Arrays & Hashing 130
2. Prefix Sum & Difference Array 236
3. Cyclic Sort 283
4. Matrix & Grid 308
5. Strings & Pattern Matching 343

Two Pointers & Sliding Window

6. Two Pointers 422
7. Sliding Window 507

Searching & Sorting

8. Binary Search 574
9. Sorting & Quickselect 661

Linked Lists

10. Linked List 695
11. Fast & Slow Pointers 766

Stacks & Queues

12. Stack 796
13. Monotonic Stack & Queue 878

Trees

14. Binary Trees (DFS & BFS) 950
15. Binary Search Trees 1068
16. Tries 1106

Heaps

17. Heaps: Top-K, Two Heaps, K-way Merge 1142

Graphs

18. Graph BFS & DFS 1220
19. Topological Sort 1316
20. Union-Find 1353
21. Shortest Paths & MST 1387

Recursion & Backtracking

ProServIQ

Tip Say the brute force in one sentence before you optimise. It proves you understand the problem.

22. Backtracking 1431

Dynamic Programming

23. 1-D Dynamic Programming 1516
24. 2-D Dynamic Programming 1592
25. Knapsack & Interval DP 1663

Greedy & Intervals

26. Intervals 1707
27. Greedy 1753

Math & Bits

28. Bit Manipulation 1813
29. Math & Geometry 1854

Design

30. Data-Structure Design 1901

Applied & AI-Era Coding

31. NumPy & ML Coding 1947
32. Pandas & Data Engineering 2005
33. Embeddings & Retrieval 2053
34. APIs & Backend Patterns 2098
35. Async & Concurrency 2148
36. AI Agent Patterns 2197

At the back

Mixed practice: spot the pattern 2249
Six small projects 2254
Answers: practice and projects 2262
Problem index & tracker 2266

ProServIQ

Before you start

Coding interviews look endless from the outside: thousands of problems, new ones every week, and a timer running. Look closer and they are not endless at all. The same few dozen ideas come back again and again, dressed up in new stories. A pair that adds up to a target. A window that slides along a list. A queue that spreads out through a maze. Once you can see the idea under the story, a "new" problem stops being new.

That is what this book is for. Not to make you memorise 300 answers, but to help you recognise the 36 patterns behind them, and to explain each one the way a patient friend would: slowly, with small examples, and with the reason it works, not just the code.

How I made it

When an idea sparks or I get stuck on a problem, I research it properly and build the finished thing, not a demo. This book was built the same way.

- **The problems** were chosen from the ones candidates report most often from big-tech interviews, then grouped by the idea that solves them.
- **Every explanation, example and drawing** was written for this book, in plain words.
- **Every Python solution was run against its tests** (Python 3.14) before its page was printed, and every code box was checked to include the imports it needs.
- **The pages were checked chapter by chapter**, for mistakes in the text and the layout. Found one anyway? Tell me and it goes into the next edition.

How to read it

The next pages help you choose one of three routes through the book. If DSA is new to you, start with the Foundations part: it explains what a hash map, a stack or a graph actually is, and where you meet them in real software, before any interview problem asks you to use one. If you already know the basics, go straight to the patterns and try each problem for twenty minutes before you read the answer.



Tip Run your idea on a tiny example out loud before you code. Interviewers love watching you test your own plan.

ProSenIQ

You will also meet a small crew at the foot of every page, each one an old symbol of wisdom or good luck:

- **Bit**, a curious robot, and **Lucky**, a waving cat, bring true stories from computing history.
- **Hoot** the owl gives you interview tips.
- **Ellie** the elephant, who never forgets, tells you what to remember.
- **Shelly** the turtle, slow and steady, shares Python tricks.



Bit Lucky Hoot Ellie Shelly

Read them when your brain needs a breather.

What this book cannot do

It cannot do the practice for you. Reading a solution feels like understanding it; solving the problem again three days later, without looking, is what actually sticks. The tracker at the back is there for exactly that.

Good luck. You are closer than you think.

Ravi Teja K, ProServIQ

ProServIQ

Three ways through this book

Nobody needs to read this book from the first page to the last. Pick the route that fits where you are today. Each route has its own page, which tells you what to read, in what order, and how to know you are ready for the next step. Every step there is a link.

Don't let the page count scare you

This book runs to about 2,300 pages, and you will never need them all at once.

- **The pages are roomy.** About 190 words each, roughly half an ordinary textbook page: the handwriting, the wide lines and the drawings take space.
- **Your first route is short.** The 57-problem Interview core is about 300 pages, some 16 a day for under three weeks. Foundations is about 80 pages.
- **One problem is one sitting.** Five or six pages and 20 to 30 minutes, with a clear start and a clear end.
- **It is really 36 small notebooks.** Every chapter also comes as its own PDF of about 60 pages, with its own answers.

What you need before page 1

Basic Python: variables, `if`, `for` and `while` loops, functions, lists and dicts, and running a `.py` file. That is all. No maths beyond school algebra, and no computer-science degree.

ProServIQ

Pick your route

Route	Take it if	It starts at
Route A: Foundations first (p. 15)	DSA is new to you, or "hash map" and "O(n log n)" still feel vague	What DSA Is, and Why Interviews Ask It (p. 22)
Route B: Interview core (p. 17)	You know the basic structures and have an interview coming	The interview framework (p. 104)
Route C: Applied (AI-era) (p. 20)	You are aiming at ML, data, backend or AI-agent roles	Interviews in the AI era (p. 122)

Not sure? Try Contains Duplicate (p. 134) without reading its solution. If you cannot write any working code in 20 minutes, take route A. If the set-based answer comes to you quickly, take route B.

How the book checks you

Reading a solution feels like understanding it. These make sure it is:

- **Predict first**, near the top of each problem: one question to answer before you read on. The next sections tell you if you were right.
- **Your turn**, after each dry run: run the method by hand on a new input. Its answer is at the end of the chapter: the **→ answer** link takes you there, and the ↩ link beside the answer brings you back.
- **A twist to try**, at the end of the Variations: the same problem with one rule changed. Adapt the solution yourself, then follow its answer link.
- **A checkpoint** at the end of each chapter: a few short questions on the whole pattern, with the answers on the pages after it.
- **Mixed practice** and **timed rounds** at the end of the book: problems with no chapter label, so you must spot the pattern yourself, as in a real interview.

ProServiQ



Fun fact Jon Bentley found that most professional programmers he tested could not write a bug-free binary search.

Route A: Foundations first

For you if DSA is new, or if words like "hash map", "recursion" or "O(n log n)" still feel vague. You need basic Python and nothing else. Plan on about two weeks for the Foundations at one topic a day, then carry on with the Interview core.

After this route you can

- explain what each core data structure is, what its operations cost and where real software uses it
- read any later chapter without meeting a word you have not seen before
- start the Interview core problems and follow every step of their solutions

Step 1: the 13 Foundations topics, in this order

Each topic ends with three "Try it" questions. Their answers are in Answers: Foundations (p. 101), after the last topic.

1. What DSA Is, and Why Interviews Ask It (p. 22)
2. Big-O Without Fear (p. 28)
3. Memory, Variables and References (p. 35)
4. Arrays and Python Lists (p. 41)
5. Strings: Text as a Row of Characters (p. 47)
6. Hash Maps: Finding Things Instantly (p. 53)
7. Linked Lists: a Treasure Hunt of Pointers (p. 59)
8. Stacks and Queues (p. 65)
9. Recursion: Solving Smaller Copies (p. 70)
10. Sorting and Searching: Order Makes Things Fast (p. 76)
11. Trees: Things Inside Things (p. 82)
12. Heaps and Priority Queues (p. 88)
13. Graphs: Maps of Connections (p. 95)

Step 2: the Interview core

Go on to Route B: Interview core (p. 17). Its first problem, Contains Duplicate (p. 134), uses exactly what the Hash Maps topic taught.

PrepEmIQ



Remember Recursion is not free: d levels deep costs O(d) memory on the call stack.

You are ready for step 2 when

You can answer each topic's "Try it" questions without looking back, and you can say in one sentence what a hash map, a stack and a queue are each good for.

ProServIQ

Route B: Interview core

For you if you already know arrays, dicts, stacks and trees, and you have an interview coming. The core is 57 problems that unlock the most others: most are on the interview lists of many big-tech companies, and every pattern from chapter 1 to chapter 30 is covered at least once. At three a day, it takes under three weeks.

After this route you can

- name the pattern behind a new problem from the words it uses, and say why out loud
- write the template of each of the 30 classic patterns from memory
- solve a core problem you have not seen for a week within its target time

Step 1: the interview toolkit, in one sitting

- The interview framework (p. 104): the six steps to follow out loud in every interview
- Which pattern is this? (p. 107): from the words in a problem to the pattern that solves it
- Big-O and Python at a glance (p. 111): the costs to quote, and the Python tools behind them
- Online Assessments: What to Expect (p. 119): read this if your first round is an online test

Step 2: the 57 core problems, in this order

Read each chapter's intro page before its first core problem. Within a chapter, each problem builds on the one before.

- **1 Arrays & Hashing:** Contains Duplicate (p. 134), Valid Anagram (p. 139), Two Sum (p. 149), Group Anagrams (p. 164), Longest Consecutive Sequence (p. 175), Product of Array Except Self (p. 180), Best Time to Buy and Sell Stock (p. 210)
- **2 Prefix Sum:** Subarray Sum Equals K (p. 249)
- **3 Cyclic Sort:** Missing Number (p. 286)
- **4 Matrix & Grid:** Spiral Matrix (p. 311)



Python ints never overflow, so (lo + hi) // 2 is safe. In Java or C++, write lo + (hi - lo) / 2.

ProSenIQ

- **5 Strings:** Longest Palindromic Substring (p. 395)
- **6 Two Pointers:** Valid Palindrome (p. 431), [[3sum]], Container With Most Water (p. 466)
- **7 Sliding Window:** Longest Substring Without Repeating Characters (p. 530), Longest Repeating Character Replacement (p. 540), Minimum Window Substring (p. 560)
- **8 Binary Search:** Search in Rotated Sorted Array (p. 613), Koko Eating Bananas (p. 628)
- **9 Sorting & Quickselect:** Kth Largest Element in an Array (p. 677)
- **10 Linked List:** Reverse Linked List (p. 699), Merge Two Sorted Lists (p. 710), Remove Nth Node From End of List (p. 720)
- **11 Fast & Slow Pointers:** Linked List Cycle (p. 779)
- **12 Stack:** Valid Parentheses (p. 800), Min Stack (p. 816)
- **13 Monotonic Stack:** Daily Temperatures (p. 888)
- **14 Binary Trees:** Maximum Depth of Binary Tree (p. 955), Invert Binary Tree (p. 960), Binary Tree Level Order Traversal (p. 1005), Lowest Common Ancestor of a Binary Tree (p. 1036)
- **15 Binary Search Trees:** Validate Binary Search Tree (p. 1081)
- **16 Tries:** Implement Trie (Prefix Tree) (p. 1110)
- **17 Heaps:** Top K Frequent Elements (p. 1161), Merge k Sorted Lists (p. 1180)
- **18 Graph BFS & DFS:** Number of Islands (p. 1230), Rotting Oranges (p. 1253)
- **19 Topological Sort:** Course Schedule (p. 1320)
- **20 Union-Find:** Number of Provinces (p. 1357)
- **21 Shortest Paths:** Network Delay Time (p. 1392)
- **22 Backtracking:** Subsets (p. 1435), Combination Sum (p. 1455), Word Search (p. 1494)
- **23 1-D Dynamic Programming:** Climbing Stairs (p. 1520), House Robber (p. 1531), Coin Change (p. 1556), Word Break (p. 1577), Longest Increasing Subsequence (p. 1582)

ProCoding



Tip "Subarray" or "substring" plus a condition is usually a sliding window.

- **24 2-D Dynamic Programming:** Unique Paths (p. 1596), Longest Common Subsequence (p. 1621)
- **25 Knapsack & Interval DP:** Partition Equal Subset Sum (p. 1667)
- **26 Intervals:** Merge Intervals (p. 1710), Non-overlapping Intervals (p. 1726)
- **27 Greedy:** Jump Game (p. 1767)
- **28 Bit Manipulation:** Single Number (p. 1817)
- **29 Math & Geometry:** Pow(x, n) (p. 1882)
- **30 Data-Structure Design:** LRU Cache (p. 1932)

Step 3: the rest of the book

The 8-week plan (p. 114) turns the remaining problems into an 8-week plan, with a 2-week version for when the interview is close.

You are ready to move on when

You solve a core problem you have not seen for a week within its target time, and you say the pattern out loud before you write any code.

ProServiQ

Route C: Applied (AI-era)

For you if you are aiming at ML, data, backend or AI-agent roles. Applied interviews still rest on the classic patterns, so this route takes the core problems those chapters build on, then the six applied chapters.

After this route you can

- name the classic pattern under an applied problem before you read its solution
- write clean, tested NumPy, pandas, async and API code under interview conditions
- check an AI assistant's code against the red flags interviewers look for

Step 1: how AI-era rounds work

Read Interviews in the AI era (p. 122) first: how AI-enabled rounds run, how to work with the assistant, and what to check in its code.

Step 2: the classic core underneath (17 problems)

- **1 Arrays & Hashing:** Contains Duplicate (p. 134), Valid Anagram (p. 139), Two Sum (p. 149), Group Anagrams (p. 164), Longest Consecutive Sequence (p. 175), Product of Array Except Self (p. 180), Best Time to Buy and Sell Stock (p. 210)
- **7 Sliding Window:** Longest Substring Without Repeating Characters (p. 530), Longest Repeating Character Replacement (p. 540), Minimum Window Substring (p. 560)
- **12 Stack:** Valid Parentheses (p. 800), Min Stack (p. 816)
- **17 Heaps:** Top K Frequent Elements (p. 1161), Merge k Sorted Lists (p. 1180)
- **18 Graph BFS & DFS:** Number of Islands (p. 1230), Rotting Oranges (p. 1253)
- **19 Topological Sort:** Course Schedule (p. 1320)

Step 3: the six applied chapters

1. 31. NumPy & ML Coding (p. 1947)
2. 32. Pandas & Data Engineering (p. 2005)
3. 33. Embeddings & Retrieval (p. 2053)

ProSavvQ


Remember A heap shows its smallest item in O(1) and removes it in O(log n).

4. 34. APIs & Backend Patterns (p. 2098)
5. 35. Async & Concurrency (p. 2148)
6. 36. AI Agent Patterns (p. 2197)

Do each problem once without an AI assistant, then again with one, checking its code against the red flags in step 1.

You are ready when

You can name the classic pattern under each applied problem before reading it, and explain where the same code would run in a production system.

ProServIQ

Hash Maps: Finding Things Instantly

Open the contacts on your phone and type "Mum". Her number appears at once. Your phone did not read every contact, one by one, until it found her: it jumped straight to the right place. That jump is what a **hash map** does, and it is the single most useful tool in coding interviews and in real software.

After this topic you can

- explain how a key jumps straight to its locker
- choose a dict, a set or a list for a lookup question
- tell which values can be keys, and turn a list into one

What it is

A hash map stores **pairs**. Each pair has a **key**, the thing you search by, and a **value**, the thing you want back.

Key	Value	Where you meet it
"Mum"	her phone number	your phone's contacts
"milk"	45, its price	a shop's price list
a user's id	that user's profile	logging in to a website

You give the map a key, and it hands back the value. It takes about the same time whether the map holds ten pairs or ten million. In Python the hash map is the `dict`, and its key-only cousin is the `set`:

A dict in four lines

```
prices = {"milk": 45, "eggs": 60}   # key -> value
prices["rice"] = 80   # add a pair
milk = prices["milk"]   # look up a key: 45
has_tea = "tea" in prices   # is the key there? False
```



Tip When the interviewer gives a hint, take it and say thanks. Ignoring a hint costs more than needing one.

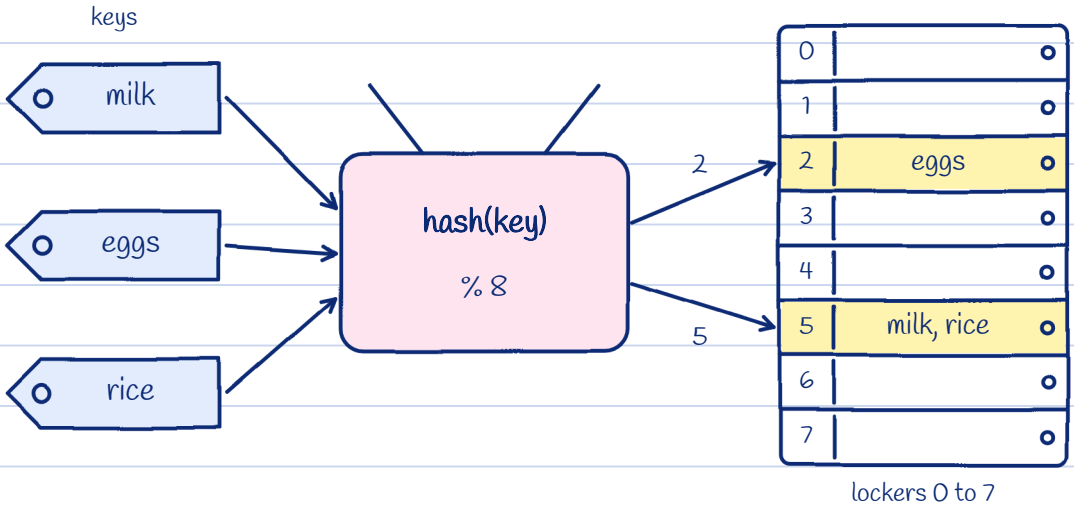
Why we need it

Without a hash map, finding something in a list means checking the items one by one until you hit it. That is O(n): for a million items, up to a million checks, every single time you search. A hash map turns that into one check, on average: O(1). Many interview problems become easy the moment you think, "What if I could remember things and find them again instantly?"

How it works: the locker room

Imagine a wall of numbered lockers.

1. To store "milk", put the word through a **hash function**: a fixed recipe that turns any key into a number. The same key always gives the same number. Take the remainder by the number of lockers, and "milk" might land on locker 5. Its value goes in locker 5.
2. To find "milk" later, run the same recipe, get 5 again, and open locker 5. No searching at all.
3. Sometimes two keys land on the same locker. That is a **collision**. The locker then holds a short list, and we check the two or three items in it. A good hash function spreads keys evenly, so these lists stay tiny.
4. When the lockers get too full, the map builds a bigger wall, about twice the size, and moves everything across. That move is slow, but it happens so rarely that adding an item still costs O(1) on average. (Python's `dict` grows when it is about two-thirds full.)



for example, "eggs" lands on locker 2, while "milk" and "rice" both land on 5: a collision

Python Since Python 3.7, a dict is guaranteed to remember the order in which keys were inserted.

Build a tiny one yourself

Here is a real, working hash map in about twenty-five lines, made from a list of lockers (programmers call them **buckets**). Python's own `dict` is written in C and handles collisions differently: instead of keeping a list in a crowded locker, it tries another free locker nearby (programmers call that **probing**). Two ways to build the same thing; the idea of jumping straight to a locker is exactly the same.

A tiny hash map: buckets, collisions, growing

```
class TinyMap:
    """A hash map made of buckets (lockers), to show how a dict works."""

    def __init__(self, size=8):
        self.buckets = [[] for _ in range(size)]    # a locker: a short list
        self.count = 0

    def _locker(self, key):
        return self.buckets[hash(key) % len(self.buckets)]

    def put(self, key, value):
        locker = self._locker(key)
        for pair in locker:
            if pair[0] == key:    # the key is already here: change its value
                pair[1] = value
                return
        locker.append([key, value])    # a collision makes the list longer
        self.count += 1
        if self.count > len(self.buckets) * 2 // 3:    # about two-thirds full
            self._grow()

    def get(self, key, default=None):
        for k, v in self._locker(key):
            if k == key:
                return v
        return default
```

ProServIQ

Fun fact Quicksort's worst case is $O(n^2)$, so many libraries switch to heapsort when quicksort recurses too deep (introsort, 1997).

```
... continued
    def _grow(self):
        pairs = [p for locker in self.buckets for p in locker]
        self.buckets = [[] for _ in range(2 * len(self.buckets))]
        for k, v in pairs:    # every key moves to its locker in the new wall
            self._locker(k).append([k, v])
```

What it costs

Operation	In Python	Usually	Worst case
Add or change a pair	`d[k] = v`	O(1)	O(n), when it grows
Look up a key	`d[k]`, `d.get(k)`	O(1)	O(n)
Is the key there?	`k in d`	O(1)	O(n)
Remove a pair	`del d[k]`	O(1)	O(n)
Visit every pair	`for k, v in d.items()`	O(n)	O(n)

Two different things can make one operation slow. **Growing**: the one insert that finds the wall full must move every pair, which is O(n), but it happens so rarely that inserts still average O(1) ("amortised"). **Collisions**: if very many keys landed in the same locker, every lookup would crawl through them. Python mixes a random value into the hashes of strings each time it starts, so nobody can easily feed it keys that all collide on purpose.

What can be a key

A key must never change while it sits in the map, or the map would look in the wrong locker. So keys must be **hashable**: numbers and strings are fine, and so are tuples whose items are hashable too (`(1, "a")` works, `(1, [2])` does not); lists, dicts and sets are not, because they can change. Try `{[1, 2]: "x"}` and Python answers `TypeError: unhashable type: 'list'`. Need a list as a key? Turn it into a tuple first: `(1, 2)`.

Where hash maps work in real software

- **Caches.** Websites remember answers they have already worked out, in a map from the question to the answer, so the next visitor gets the answer instantly. Popular cache servers such as Redis and Memcached are, at heart, huge hash maps kept in memory.
- **Databases.** To match rows from two tables, a database often builds a hash map of one table and looks up each row of the other: a **hash join**. Many databases also offer hash indexes for "find exactly this value" searches.
- **Programming languages.** Python keeps every module's names, and the attributes of most objects, in dicts. Writing `obj.name` usually means a dictionary lookup behind the scenes.
- **Counting and de-duplicating.** Counting words in a log file, spotting a payment that was charged twice, counting unique visitors: each is a "have I seen this before?" question, answered by a map or a set.
- **Version control.** Git stores every version of every file under a name made by hashing the file's content, and finds it again by that name: a key-value store built on a hash function.

Why it belongs to DSA

DSA means **data structures** (ways to organise data so that certain questions become fast) and **algorithms** (step-by-step methods that answer those questions). A hash map is a data structure built to answer one question fast: "what goes with this key?". Many algorithms in this book stand on that one question:

- "Have I seen this before?" → Two Sum (p. 149), Contains Duplicate (p. 134)
- "How many of each?" → Valid Anagram (p. 139), Top K Frequent Elements (p. 1161)
- "Which group does this belong to?" → Group Anagrams (p. 164)
- "Did I work this out already?" → memoised dynamic programming, and LRU Cache (p. 1932)



Python The default recursion limit is 1,000. Deep recursion on big inputs needs a loop or an explicit stack.

When not to use one

- **You need order.** "Smallest first", "everything between 10 and 20", "the next bigger value": a hash map cannot answer those quickly. Python's `dict` does remember the order in which keys were *inserted*, but it never keeps them *sorted*. Use a sorted list with binary search, a heap or a balanced tree.
- **Memory is tight.** A map uses several times the memory of a plain list holding the same items, because of the empty lockers and the stored hashes.
- **The data is tiny.** For five items, scanning a list is just as quick and simpler to read.

Try it

Three quick questions. Answer them on paper first; each one links to its answer, after the last Foundations topic.

1. Which would you use? (a) Count how often each word appears. (b) Remember only which words you have seen. (c) Keep scores sorted, best first. → answer, p. 102
2. A dict holds a million pairs. Roughly what does `k in d` cost? And the rare step when the dict grows? → answer, p. 102
3. What does this print? → answer, p. 102

```
d = {}
for w in ["tea", "jam", "tea"]:
    d[w] = d.get(w, 0) + 1
print(d, (1, 2) in {(1, 2): "x"})
```

Remember

- A hash map answers "what goes with this key?" in O(1) on average.
- Keys must not change: use strings, numbers, or tuples of those.
- When a problem says "have I seen…", "how many…" or "group by…", reach for a `dict` or a `set` first.

ProSenIQ



Fun fact Alan Turing described a stack for subroutine calls in 1946, calling its operations "bury" and "unbury".

1. Arrays & Hashing

Chapter 1 of 36 · Arrays, Strings & Hashing · 19 problems, about 4.5 hours at their target times

Think of a cloakroom attendant at a theatre. You hand over your coat and get a numbered ticket. At the end of the night you show the ticket, and the attendant walks straight to the right hook. Nobody searches every coat. That jump, from "what I know" straight to "where it is", is what a **hash map** gives you in code: a Python `dict` (key → value) or `set` (keys only). It turns the question "have I seen this before?" into a single step, O(1) on average. Most problems in this chapter take $O(n^2)$ with two loops and O(n) once a hash map remembers what the first loop would have searched for.

After this chapter you can

- answer "have I seen this before?" in one pass with a `set` or a `dict`, instead of a second loop
- choose a key that makes equal things land together: the value, its sorted letters, a tuple of counts
- replace an inner loop with a running value, such as the lowest price so far

When you see this

- "Find two items that...", "is there a repeat?", "count how many times..."
- "Group the things that are the same under some rule" (anagrams, same letters, same shape)
- You need to find something by its **value**, not by its position
- You need a tally of counts, or a quick "is it in here?" check

Core idea

- Trade memory for time: write down what you have seen in a `dict` or `set`, then ask it instead of searching again.
- Choose the **key** with care. The key is whatever makes two items "the same" for this problem: the value itself, the sorted letters, a tuple of counts.
- One pass is usually enough. Look in the map first, then write the current item in. That order stops an item from matching itself.



Tip "Count the ways" or "minimum cost" with choices that repeat is usually dynamic programming.

ProSenio



Template

Hash map, one pass

```
def pattern(items):
    seen = {}    # key -> what we need later (index, count, list)
    for i, x in enumerate(items):
        key = make_key(x)    # x itself, sorted(x), or a tuple of counts
        if key in seen:
            return ...    # found a match: use seen[key]
        seen[key] = i
    return ...    # nothing matched
```

Not every problem here needs a map. A few use a cousin of the same idea: keep a **running value** (the lowest price so far, the product of everything to the left) so that one pass replaces an inner loop. Best Time to Buy and Sell Stock and Product of Array Except Self work that way.

Python tools you will use

Tool	What it does	Cost
`x in a_set`, `x in a_dict`	is x stored here?	O(1) on average
`d.get(k, 0)`	the value, or a default when k is missing	O(1)
`collections.Counter(xs)`	a tally: each item → how many times	O(n)
`collections.defaultdict(list)`	a map whose missing keys start as empty lists	O(1) per use
`tuple(xs)`, `"".join(sorted(s))`	turn a list or a word into a key	O(k) or O(k log k)

Common mistakes

- Writing into the map **before** looking, so an item matches itself.
- Using a list as a key. Lists can change, so Python refuses them; use a tuple.



Tip Two Sum, Contains Duplicate and Group Anagrams ask the same question in disguise: have I seen this before?

- Forgetting that `in` on a **list** reads it item by item, O(n). Only sets and dicts answer in O(1).
- Sorting the list to make lookups easy, then losing the original positions the problem asked for.

From my notebook

- Before any clever idea, ask: would a dictionary of what I have already seen make this one pass? Half this chapter says yes.
- Say the key out loud: what am I looking up, and what do I store with it? Most bugs here are a wrong key.

Problems in this chapter

#	Problem	LeetCode	Level	Priority	Page
1	Contains Duplicate	#217	Easy	★★★	134
2	Valid Anagram	#242	Easy	★★★	139
3	First Unique Character in a String	#387	Easy	★★	144
4	Two Sum	#1	Easy	★★★	149
5	Intersection of Two Arrays	#349	Easy	★★	154
6	Majority Element	#169	Easy	★★	159
7	Group Anagrams	#49	Medium	★★★	164
8	Valid Sudoku	#36	Medium	★★	169
9	Longest Consecutive Sequence	#128	Medium	★★★	175
10	Product of Array Except Self	#238	Medium	★★★	180
11	Ransom Note	#383	Easy	★	185
12	Isomorphic Strings	#205	Easy	★★	190
13	Word Pattern	#290	Easy	★	195
14	Happy Number	#202	Easy	★	200
15	Sort Characters By Frequency	#451	Medium	★	205
16	Best Time to Buy and Sell Stock	#121	Easy	★★★	210
17	Maximum Product of Three Numbers	#628	Easy	★	215

ProSenio

Fun fact Paul Bachmann used big-O notation in 1894, and Donald Knuth made it standard for algorithms in the 1970s.

#	Problem	LeetCode	Level	Priority	Page
18	Pascal's Triangle	#118	Easy	★	221
19	Rotate Array	#189	Medium	★	226

ProServIQ

Problem 1 · Contains Duplicate

LeetCode #217 · Easy · ★★★ Must-do · Target: 10 min

On interview lists at many big-tech companies, including Adobe and Visa

Practise it: search "contains duplicate" or "check if array has duplicates" on LeetCode, HackerRank or Codility

Problem

You are handing out raffle tickets at a school fair, and each ticket should carry a different number. At the end of the day you have the list of numbers you wrote down. Did you write any number twice?

You get a list of whole numbers `nums`. Return `True` if some value appears at least twice, and `False` if every value is different.

Example 1

`nums = [14, 3, 9, 27, 3]` → **True**. The number 3 was handed out twice.

Example 2

`nums = [8, -2, 0, 15]` → **False**. All four numbers are different.

Clarify first

- Can the list be empty? (Yes. Then nothing repeats, so the answer is `False`.)
- Do they want **which** number repeats? (No, only whether one does. So we can stop at the first repeat.)
- May I change the list, for example by sorting it? (Ask. If not, sort a copy.)

Predict first

Before reading on: can you answer in **one** walk along the list? If so, what must you remember as you walk?

ProSenIQ

Brute force: compare every pair

- Take the first number and compare it with every number after it. Then take the second, and so on.
- With n numbers that is about $n^2/2$ comparisons. Fine for a few hundred tickets; 10^5 tickets means 5×10^9 checks.

Brute force

```
def contains_duplicate_brute(nums: list[int]) -> bool:
    n = len(nums)
    for i in range(n):
        for j in range(i + 1, n):
            if nums[i] == nums[j]:
                return True
    return False
```

- Time $O(n^2)$, space $O(1)$.

Better: sort, then look at neighbours

- Sort the numbers and equal values end up side by side. One walk along the neighbours then finds a repeat.

Better: sort and compare neighbours

```
def contains_duplicate_sorted(nums: list[int]) -> bool:
    nums = sorted(nums)    # a copy, so the caller's list is untouched
    for i in range(1, len(nums)):
        if nums[i] == nums[i - 1]:    # equal values sit side by side now
            return True
    return False
```

- Time $O(n \log n)$, space $O(n)$ for the sorted copy. Sorting the list itself with `nums.sort()` saves the copy, but Python's sort still borrows scratch memory for up to $n/2$ items, so the extra space stays $O(n)$.

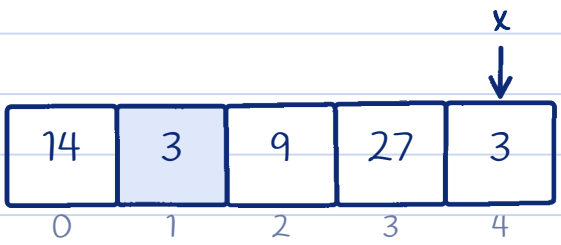
Sorting still does far more than we asked for. We do not need the numbers in order; we only need to recognise one we have met before. What remembers things instantly?

Fun fact Since Python 3.6, dicts store their entries in a compact array, which made them noticeably smaller.

ProServIQ

Optimised: a set of the numbers we have seen

- Walk the list once and keep a **set**: a bag of values that answers "is this in here?" in one step, on average, however big it gets.
- For each number, ask the set first. If the number is there, we have found a repeat. If not, drop it in and move on.
- We stop at the first repeat, so `[5, 5, ...]` is answered after two steps.



at the last ticket: seen = {14, 3, 9, 27}, and 3 is already in it → True

Pseudocode

1. seen = empty set
2. for each number x in nums:
3. if x is in seen: return True
4. add x to seen
5. return False

Optimised

```
def contains_duplicate(nums: list[int]) -> bool:
    seen = set()
    for x in nums:
        if x in seen:
            return True    # stop at the first repeat
        seen.add(x)
    return False
```

- A one-line version: `len(set(nums)) < len(nums)`. Same O(n), but it always reads the whole list and builds the whole set, even when the repeat is at the front.

Python Strings cannot change, so build a long string with "".join(parts), not += in a loop.

ProServIQ

Why it works

- At every step, `seen` holds exactly the numbers before the current one. So "x is in seen" means "x appeared earlier", which is a repeat.
- If a value appears twice, its second copy is checked after the first copy went into the set, so the repeat is always caught.
- The loop visits each number once and either returns early or reaches the end, where no repeat exists, so `False` is correct.

Dry run: nums = [14, 3, 9, 27, 3]

x	x in seen?	seen after this step
14	no	{14}
3	no	{14, 3}
9	no	{14, 3, 9}
27	no	{14, 3, 9, 27}
3	**yes** → return True	

Your turn

Run the set version by hand on `nums = [11, 6, 0, 6, 11]`. At which position does it stop, and what is in `seen` at that moment? → answer, p. 232

Complexity

Approach	Time	Space	Why
Every pair	$O(n^2)$	$O(1)$	about $n^2/2$ comparisons
Sort + neighbours	$O(n \log n)$	$O(n)$	the sort dominates; even sorting in place, Python's sort uses up to $n/2$ scratch slots
Hash set	$O(n)$	$O(n)$	one pass; the set may hold all n numbers

Edge cases

- Empty list or one number: nothing can repeat → `False`. ✓
- Negative numbers and zero: a set treats them like any other value. ✓
- The repeat is the first and last number: found on the final step. ✓
- Every number the same: found on the second step. ✓

Remember Sorting a word's letters gives every anagram the same key: "listen" and "silent" both become "eilnst".

ProServiQ

Common mistakes

- Writing `if x in nums[:i]`. `in` on a **list** checks items one by one, so the "single pass" is secretly $O(n^2)$.
- Sorting the caller's list in place when you were told not to change the input.
- Returning the repeated number (or `1`) when a `True` / `False` answer was asked for.

Variations

- **Contains Duplicate II (LC 219, p. 515)**: a repeat no more than k positions apart → keep a set of only the last k numbers (a sliding window).
- **Find the Duplicate Number (LC 287, p. 789)**: no extra memory and no changes allowed → fast and slow pointers.
- **Find All Duplicates in an Array (LC 442, p. 296)**: values from 1 to n → mark each value's home position instead of using a set (the cyclic-sort family).
- **A twist to try**: return `True` only if some value appears at least **three** times. What replaces the set? → answer, p. 232

Interview takeaways

- This is the smallest "have I seen it?" problem. Give all three ideas with their costs: pairs $O(n^2)$, sort $O(n \log n)$, set $O(n)$.
- A likely follow-up: "memory is tight". Sorting the list itself avoids the set's n entries, though Python's sort still needs up to n/2 scratch slots; only a hand-written sort such as heapsort works in $O(1)$ extra space.
- The same seen-set shape returns in Happy Number (LC 202, p. 200) and Longest Consecutive Sequence (LC 128, p. 175).

ProSenIQ

Problem 2 · Valid Anagram

LeetCode #242 · Easy · ★★★ Must-do · Target: 10 min

On interview lists at many big-tech companies, including Adobe, Visa, Amazon and Apple

Practise it: search "valid anagram" or "check if two strings are anagrams" on LeetCode, HackerRank or Codility

Problem

You have a bag of letter tiles that spells `night`. Can you lay out the word `thing` with exactly those tiles, using every tile and needing no extra one?

Two words like that are **anagrams**: the same letters, each the same number of times, only in a different order. You get two strings `s` and `t` of lowercase English letters. Return `True` if `t` is an anagram of `s`, and `False` otherwise.

Example 1

`s = "night"`, `t = "thing"` → **True.** Both use one each of n, i, g, h, t.

Example 2

`s = "bell"`, `t = "belt"` → **False.** `t` needs a second t tile, and the second l tile is left over.

Clarify first

- Only lowercase `a – z`? (Yes. That lets a list of 26 counters replace a dictionary. With other alphabets, use a `Counter`.)
- Do spaces or capitals count? (Here every character counts exactly as it is.)
- Can a string be empty? (Yes. Two empty strings are anagrams of each other.)

Predict first

For two long words, which will be faster: sorting both and comparing, or counting the letters? Decide before you read on, and say why.

Tip When a problem demands O(n) time, a hash map or a 26-slot counting array is usually the trick.

Brute force: sort both strings

- Sort the letters of each word. Anagrams turn into the same string: `night` and `thing` both become `ghint`.

Brute force

```
def is_anagram_sorted(s: str, t: str) -> bool:
    return sorted(s) == sorted(t)    # anagrams sort to the same string
```

- Time O(n log n) for the two sorts, space O(n) for the sorted copies. Short, correct, and a fine first answer.

But think about the tiles again. To check them, would you line them up alphabetically? Or would you just count them?

Optimised: one tally of 26 counters

- We do not need the letters in order, only **how many** of each. Counting takes one pass: O(n).
- One list of 26 counters is enough, one per letter. Each letter of `s` adds 1 to its counter; each letter of `t` takes 1 away. If every counter ends at 0, the two words used the same tiles.
- Check the lengths first. Different lengths can never be anagrams, and equal lengths let us walk both strings side by side with `zip`.

b	+1 (from s) − 1 (from t) = 0
e	+1 − 1 = 0
l	+2 − 1 = 1
t	− 1 = −1

s = "bell", t = "belt": two counters are not 0 → False

ProGenQ

Pseudocode

1. if the lengths differ: return False
2. count = 26 zeros
3. for each pair of letters (a from s, b from t):
4. count[a] += 1
5. count[b] -= 1
6. return True if every count is 0, else False

Optimised

```
def is_anagram(s: str, t: str) -> bool:
    if len(s) != len(t):
        return False    # different lengths can never match
    count = [0] * 26
    for a, b in zip(s, t):
        count[ord(a) - ord("a")] += 1    # s adds a letter ...
        count[ord(b) - ord("a")] -= 1    # ... t takes one away
    return all(c == 0 for c in count)
```

- For any alphabet, compare two `Counter`s (dictionaries of character counts). Same O(n):

Follow-up: any alphabet

```
from collections import Counter


def is_anagram_counter(s: str, t: str) -> bool:
    return Counter(s) == Counter(t)    # any alphabet, Unicode included
```

Why it works

- After each step, counter `c` holds (copies of `c` seen so far in `s`) minus (copies seen so far in `t`). At the end that is the full difference for every letter.
- All 26 differences are 0 exactly when both words have the same count of every letter, which is the definition of an anagram. Nothing is left unchecked.
- The loop runs once per letter pair, so it stops after n steps.

Python Counter(a) == Counter(b) checks whether two words are anagrams in one line.

ProServIQ

Dry run: s = "dusty", t = "study"

step	a (from s)	b (from t)	counters not at 0 after this step
1	d	s	d: 1, s: −1
2	u	t	d: 1, s: −1, u: 1, t: −1
3	s	u	d: 1, t: −1
4	t	d	none
5	y	y	none
end			all zero → True

Your turn

Run the counter version on `s = "brag"`, `t = "garb"`. Which counters are not zero after step 2, and what is returned? → answer, p. 232

Complexity

Approach	Time	Space	Why
Sort both	O(n log n)	O(n)	two sorted copies
Count list	O(n)	O(1)	26 counters, however long the words are
Counter	O(n)	O(k)	one entry per distinct character, k of them

Edge cases

- Both empty: the lengths match, the loop never runs, all counters stay 0 → `True`. ✓
- Different lengths, like `"ten"` and `"tent"`: rejected on the first line. ✓
- Same letters in different amounts, like `"pool"` and `"plop"`: the o counter ends at 1 → `False`. ✓

Common mistakes

- Comparing `set(s) == set(t)`. Sets forget how many, so `"pool"` and `"plop"` both become {p, o, l} and look equal.
- Skipping the length check while walking with `zip`, which stops at the shorter string: `"ten"` against `"tent"` would return `True`.



Fun fact Edward Fredkin named the trie in 1960, taking the word from "retrieval".

Variations

- **Group Anagrams (LC 49, p. 164)**: many words → the counts become a dictionary **key** (this chapter).
- **Ransom Note (LC 383, p. 185)**: one word must fit **inside** the other → counts as a budget, only subtract (this chapter).
- **A twist to try**: the two words may now have different lengths. Return how many tiles you must add or remove in total to make `t` an anagram of `s`. → answer, p. 232

Interview takeaways

- Say: "anagram means equal letter counts". Sorting is only the warm-up.
- Other alphabets? Swap the 26 slots for a `Counter`; space becomes O(k).
- Count-and-compare returns in Ransom Note and Minimum Window Substring.

ProServIQ


Tip Look before you write: check the map for a partner first, then store the current item.

Problem 3 · First Unique Character in a String

LeetCode #387 · Easy · ★★ Core · Target: 10 min

On interview lists at many big-tech companies, including Goldman Sachs, Adobe, Visa and Meta

Practise it: search "first unique character in a string" or "first non repeating character" on LeetCode, HackerRank or Codility

Problem

A queue of people waits at a café, each wearing a coloured badge shown as a letter. The barista wants to serve the first person whose badge colour nobody else in the queue shares. Where in the queue are they?

You get a string `s` of lowercase English letters. Return the **index** (position, counting from 0) of the first character that appears **exactly once** in `s`. If every character repeats, return `-1`.

Example 1

`s = "swiss"` → **1**. The s's repeat; `w` at index 1 appears once and comes before the `i`.

Example 2

`s = "noon"` → **-1**. Both n and o appear twice.

Clarify first

- Index or the character itself? (The index. So we must keep track of positions.)
- Only lowercase `a – z`? (Yes. That allows a 26-slot count; a `Counter` works for any alphabet.)
- What if the string is empty? (Nothing appears once → `-1`.)

Predict first

Can you decide that a letter is unique the moment you first meet it? If not, what do you need to know first?

ProServiQ

Brute force: count each character from scratch

- For each position, count how often its character appears in the whole string. The first one with a count of 1 is the answer.

Brute force

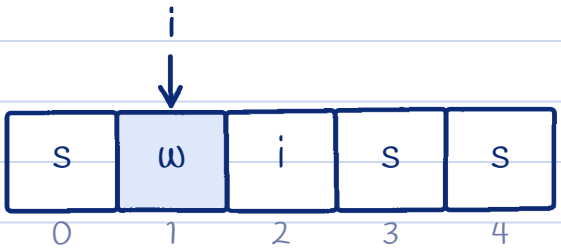
```
def first_uniq_char_brute(s: str) -> int:
    for i, ch in enumerate(s):
        if s.count(ch) == 1:    # reads the whole string every time
            return i
    return -1
```

- `s.count(ch)` reads the whole string every time: $O(n^2)$ time, $O(1)$ space. With 10^5 characters that is 10^{10} steps.

We keep asking the same question, "how many of this letter are there?", and answering it from scratch. Why not answer it once for every letter, and then just look it up?

Optimised: count once, then scan in order

- Pass 1: count every character with a **frequency table** (a dictionary from each character to how many times it appears, here a `Counter`).
- Pass 2: walk the string again, **in its original order**, and return the first index whose character has a count of 1.
- The second pass must read the string, not the table: the string remembers the order; the table only remembers how many.



counts s 3, w 1, i 1: walking in order, index 1 (w) is the first with count 1

Remember A set forgets how many times it saw something; a Counter remembers. Use a Counter when copies matter.

Pseudocode

1. count = how many times each character appears in s
2. for each index i and character ch in s:
3. if count[ch] is 1: return i
4. return -1

Optimised

```
from collections import Counter


def first_uniq_char(s: str) -> int:
    count = Counter(s)    # pass 1: how many of each character
    for i, ch in enumerate(s):    # pass 2: in the string's own order
        if count[ch] == 1:
            return i
    return -1
```

PreServIQ

Why it works

- After pass 1, count[ch] is the exact number of times ch appears in the **whole** string, so "count is 1" means "appears exactly once".
- Pass 2 tries the positions from left to right, so the first one it returns is the earliest unique character. Nothing earlier was skipped.
- If pass 2 reaches the end, no character had a count of 1, so `-1` is right. Each pass is a single loop over n characters, so both stop.

Dry run: s = "level"

Pass 1 gives l: 2, e: 2, v: 1.

i	ch	count[ch]	action
0	l	2	keep going
1	e	2	keep going
2	v	1	return 2

Python divmod(a, b) returns the quotient and the remainder together.

Your turn

Run both passes on `s = "teeter"`. What does pass 1 count, and which index does pass 2 return? → answer, p. 232

Complexity

Approach	Time	Space	Why
Count each from scratch	$O(n^2)$	$O(1)$	`s.count` reads the whole string each time
Count once, scan in order	$O(n)$	$O(1)$	two passes; at most 26 letters in the table

Edge cases

- Empty string: the loop never runs → `-1`. ✓
- One character: it appears once → `0`. ✓
- Every character repeats, as in Example 2 → `-1`. ✓
- The unique character is the last one, as in `"aab"` → `2`. ✓
- Two unique characters, as in Example 1: the earlier one wins. ✓

Common mistakes

- Looping over the Counter in pass 2 instead of the string. The answer then depends on the table's order, not the queue's, and it gives a character, not an index.
- Returning the character instead of its index.
- Using `s.index(ch) == s.rindex(ch)` for every position: correct, but each call reads the string, so it is $O(n^2)$ again.

Variations

- **Valid Anagram (LC 242, p. 139)**: the same count table, compared between two strings (this chapter).
- **First unique number in a stream**: characters arrive one at a time → keep the counts plus a queue of candidates, and drop repeated ones from its front.

Python A list cannot be a dict key, but tuple(counts) can. That is how Group Anagrams builds its key.

- **Sort Characters By Frequency (LC 451, p. 205)**: the same count table, then order by count (this chapter).
- **A twist to try**: you get two strings, `s` and `t`. Return the index in `s` of the first character that appears exactly once in `s` **and** exactly once in `t`. → answer, p. 232

Interview takeaways

- Say: "count once, then scan the string in its own order". The two-pass shape is the whole trick.
- A common follow-up is the stream version: a queue of candidates plus counts, so each new character costs O(1) on average.
- Counting first and deciding second is the shape of Valid Anagram, Ransom Note and Top K Frequent Elements too.

ProServ



Fun fact With both of its tricks, union-find runs in almost constant time per operation: the bound stays below 5 for any real input.

Answers: 1. Arrays & Hashing

1. Contains Duplicate · Your turn: It stops at index 3, the second 6, and returns `True`. At that moment `seen = {11, 6, 0}`; the second 11 is never read. ↩ p. 137

Twist: Swap the set for a dict of counts (`count[x] = count.get(x, 0) + 1`) and return `True` when a count reaches 3. Still one pass, O(n). ↩ p. 138

2. Valid Anagram · Your turn: After step 2: b: 1, g: −1, r: 1, a: −1. Steps 3 and 4 cancel them all, so it returns `True`. ↩ p. 142

Twist: Run the same +1/−1 counters over both whole words and return the sum of their absolute values (`bell`, `belt` → 2). O(n + m). ↩ p. 143

3. First Unique Character in a String · Your turn: Pass 1 counts t: 2, e: 3, r: 1. Pass 2 skips every t and e and returns **5**, the r at the very end. ↩ p. 147

Twist: Count each string separately, then scan `s` for the first index whose letter has count 1 in both (else `-1`). O(n + m). ↩ p. 147

4. Two Sum · Your turn: Just before, `seen = {3: 0, 9: 1}`. At index 2, −4 needs 9, found at index 1, so it returns **[1, 2]**. ↩ p. 152

Twist: Each number's **latest** index: overwrite it as you go. At each i, if target − x is stored, compare i minus that index with the best gap so far. One pass, O(n). ↩ p. 153

5. Intersection of Two Arrays · Your turn: `nums2` is shorter, so `seen = {3, 6, 7}`. Walking `nums1` reports 7, then 3; the repeats find nothing. It returns **[7, 3]**, leaving `{6}`. ↩ p. 157

Twist: Make a set of each list and keep the values of each set that the other lacks (`a ^ b` in Python, the symmetric difference). O(n + m). ↩ p. 158

6. Majority Element · Your turn: After index 3 the candidate is 2 with 0 votes. The 6 at index 4 takes over and keeps the lead to the end, so it returns **6**. ↩ p. 162

Twist: Vote with weights: a slip for the candidate adds w, any other subtracts w; if votes drop below 0, that slip takes over with the leftover. O(n), O(1). ↩ p. 163

7. Group Anagrams · Your turn: `tone` gets the label `enot`, the same as `note`. The groups are [note, tone], [ten, net] and [eon]: three envelopes. ↩ p. 167

That was the sample.

You have seen the routes, one Foundations topic and the first problems of chapter 1. The full book goes on for 2,288 pages:

- 340 problems in 36 patterns, each with tested Python and "why it works"
- 13 Foundations topics for beginners
- 6 AI-era chapters: NumPy, pandas, embeddings, APIs, async, agents
- a checkpoint after every chapter, mixed sets, timed coding rounds, 6 projects
- 913 answers, each one link away from its question
- the 57-problem Interview core, 8-week and 14-day plans, and a tracker

Launch price: $29 (₹699 in India)

proserviq.com/think-in-patterns