

DSA

Data Structures & Algorithms in Python

Start from Python basics, learn 20 patterns, and crack coding rounds. Every problem in simple steps: brute force, the key trick, a dry run, clean code and Big-O.

```
crack_the_interview.py

def solve(problem):
    pattern = spot_pattern(problem)      # 20 patterns
    trick = pattern.key_trick()           # the one idea
    return code(trick, lang="python")     # O(n), not O(n^2)

offer = solve("your next interview")    # ✓ Accepted
```

Two Pointers

Sliding Window

Hashing

Binary Search

Backtracking

Linked Lists

Monotonic Stack

Trees

Heaps

BFS / DFS

Dijkstra

Union-Find

Dynamic Programming

Greedy

Tries

Bit Tricks

WRITTEN BY

Shreyash Tripathi



Data Structures & Algorithms

Pattern-Based Interview Guide in Python, from basics to hard problems

1 Spot the pattern



2 Find the trick



3 Dry run



4 Code



5 Big-O

WRITTEN BY

Shreyash Tripathi

Interview Edition, 2026

© Shreyash Tripathi. All rights reserved.

BEFORE YOU BEGIN

Preface

Coding rounds scare most people for one reason: every problem looks new. It is not. Almost every interview problem is one of about twenty **patterns** wearing a different costume. Once you can see the pattern, the "hard" problem turns into a template you have already practised. This book teaches you to see those patterns.

We start from the very basics of Python in Chapter 0, so the language never gets in your way. Then each chapter takes one pattern, explains it with a simple analogy, gives you a reusable code template, and solves 6 to 9 classic interview problems the same way every time: the brute force first, then the **one key trick** that unlocks the problem, a step-by-step dry run, clean Python code, and the time and space complexity. You will also learn what interviewers ask next, and the bugs that cost people offers.

Do not just read the solutions. Cover the code, try the problem for 20 minutes, then compare. That habit, repeated for 30 days, is the whole secret.

Shreyash Tripathi

How Every Problem Is Solved in This Book

1

Spot the pattern

Signal words in the problem tell you which chapter it belongs to.

2

Brute force

Say the simple answer first and its Big-O.

3

Key trick

The one idea that removes the wasted work.

4

Dry run + code

Trace a tiny example, then write clean Python.

5

Big-O + follow-ups

State complexity and handle what comes next.

Features of Every Chapter

💡 DO YOU KNOW?

Real facts and history.

😄 FUNNY SIDE

A joke that makes the idea stick.

🔑 KEY POINT

The line to remember.

🎯 INTERVIEW CORNER

Questions interviewers really ask, with model answers.

AT WORK

How the concept shows up in a real job.

COMMON MISTAKE

Traps that cost marks and cause bugs.

CHECK YOUR PROGRESS

Short questions with answers after every topic, plus exercises and a full answer key at the end of each chapter.

Your 30-Day Plan

Days	Chapters	Goal
1 to 3	0 to 2	Language basics, problem-solving method, Big-O
4 to 12	3 to 8	Arrays, two pointers, sliding window, hashing, binary search, sorting
13 to 18	9 to 13	Backtracking, linked lists, stacks, trees, heaps
19 to 25	14 to 18	Graphs, dynamic programming, greedy
26 to 30	19 to 20	Tricks, then the 100 must-do problems against the clock

Contents

PART I: FOUNDATIONS

0	Python Basics for DSA (Start Here)	7
1	How to Approach Any DSA Problem	29
2	Time and Space Complexity	44

PART II: LINEAR PATTERNS

3	Arrays, Strings and Prefix Sums	63
4	Two Pointers	79
5	Sliding Window	96
6	Hashing	114
7	Binary Search	133
8	Sorting and Intervals	151

PART III: RECURSION AND LINEAR STRUCTURES

9	Recursion and Backtracking	171
10	Linked Lists	191
11	Stacks, Queues and Monotonic Stack	209

PART IV: TREES, HEAPS AND GRAPHS

12	Trees and Binary Search Trees	227
13	Heaps and Priority Queues	247
14	Graphs I: BFS, DFS and Topological Sort	265
15	Graphs II: Shortest Paths, Union-Find and MST	284

PART V: DYNAMIC PROGRAMMING AND GREEDY

16	Dynamic Programming I: Foundations	303
17	Dynamic Programming II: Grids, Strings and Subsequences	319
18	Greedy Algorithms	338

PART VI: TRICKS AND PRACTICE

19	Tries, Bit Manipulation and Math Tricks	356
20	Must-Do Problem Bank and Pattern Cheat Sheet	375

APPENDICES

A	Pattern Recognition Cheat Sheet	393
B	Big-O and Python Collections Cheat Sheet	396
C	Python Code Templates	399
D	Glossary	404

PART I

Foundations

Language basics, how to approach any problem, and how to measure an algorithm.

CHAPTER 0

Python Basics for DSA (Start Here)

"Python is like a well-stocked kitchen: half the cooking is already done, if you know which shelf to open."

After studying this chapter, you will be able to

- Use Python numbers, strings, lists, tuples, sets and dicts correctly in a coding round.
- Pick the right tool from `collections`, `heapq` and `bisect` without searching the docs.
- Sort with `key`, `lambda` and `cmp_to_key`, and loop cleanly with `enumerate` and `zip`.
- Write recursion with memoization and avoid the recursion-depth crash.
- State the Big-O of common Python operations from memory.
- Avoid the classic Python traps: mutable defaults, shallow copies, the `[[0]*m]*n` grid bug.

Think of a coding round like a cricket match on a short boundary ground. Java players hit hard, but they have to run every single (declare types, write getters, import ten classes). Python players can hit a six with one line: `Counter(s).most_common(1)`. That speed only helps if you know the shots.

This chapter is your net practice. We start from variables and loops, then move to the built-in tools that make Python the favourite language of many interview candidates. Read it once fully, then keep it open as a cheat sheet while solving the rest of the book.

0.1 Variables, Types and Numbers

A variable in Python is a name tag stuck on an object. You do not declare a type; the object carries its own type.

```
x = 10           # int
pi = 3.14        # float
name = "Ravi"    # str
ok = True        # bool (True/False, capital letters)
nothing = None   # like null in Java
print(type(x))   # <class 'int'>
a, b = 1, 2
a, b = b, a       # swap in one line, no temp variable
```

0.1.1 int never overflows

Python `int` has arbitrary precision. `2**100` just works. In Java you would need `long` or `BigInteger`. The trade-off: very large numbers are slower, and some problems ask you to return the answer "modulo $10^9 + 7$ ", so you still apply `% MOD` yourself.

```
MOD = 10**9 + 7
big = 2**100          # 1267650600228229401496703205376
ans = (big * big) % MOD # take mod when the problem asks
```

0.1.2 Division: /, // and % (with negatives)

`/` always gives a float. `//` is floor division: it rounds towards minus infinity, not towards zero like Java. `%` follows the same rule, so the result takes the sign of the divisor.

Expression	Python	Java
<code>7 / 2</code>	3.5	3 (int division)
<code>7 // 2</code>	3	n/a
<code>-7 // 2</code>	-4	-3 (for <code>-7 / 2</code>)
<code>-7 % 3</code>	2	-1
<code>int(-7 / 2)</code>	-3 (truncates)	n/a

Table 0.1: division rules compared

⚠ COMMON MISTAKE

Problems like "Evaluate Reverse Polish Notation" say "division truncates toward zero". `-7 // 2` gives -4, which is wrong there. Use `int(a / b)` for small values, or handle the sign yourself. Good news: Python's `%` is great for circular arrays, because `(i - 1) % n` is never negative.

0.1.3 Float pitfalls

Floats are stored in binary, so `0.1 + 0.2 == 0.3` is **False** (the sum is 0.30000000000000004). Never compare floats with `==`. Use `math.isclose(a, b)` or `abs(a - b) < 1e-9`. Better: stay in integers. To compare slopes, compare `dy1 * dx2 == dy2 * dx1` instead of dividing.

✅ CHECK YOUR PROGRESS 0.1

1. What do `-9 // 4` and `-9 % 4` return?

Answer: -3 and 3. Floor of -2.25 is -3, and $-3 \times 4 + 3 = -9$.

2. Why do you still use `% MOD` in Python if ints never overflow?

Answer: The problem asks for the answer modulo a prime, and huge numbers make arithmetic slow. Taking mod at each step keeps numbers small and fast.

0.2 Control Flow and Functions

0.2.1 if / elif / else

```
marks = 72
if marks ≥ 90:
    grade = "A"
elif marks ≥ 60:      # Python writes elif, not else if
    grade = "B"
else:
    grade = "C"
label = "pass" if marks ≥ 35 else "fail"    # one-line ternary
if 0 ≤ marks ≤ 100: print("valid")         # chained comparison
```

Indentation is the block. Mixing tabs and spaces causes errors. Use 4 spaces. Logical operators are words: `and`, `or`, `not`. Empty list, empty string, 0 and None are all "falsy", so `if not stack:` means "stack is empty".

0.2.2 for, while and range

```
for i in range(5):      # 0,1,2,3,4
    print(i)
for i in range(2, 10, 3): # 2,5,8 (start, stop excluded, step)
    print(i)
for i in range(4, -1, -1): # 4,3,2,1,0 : going backwards
    print(i)

i = 0
while i < 3:
    i += 1                # no i++ in Python
for x in [3, 1, 2]:
    if x == 1:
        break
else:
    print("no break happened") # for-else runs only if loop did not break
```

0.2.3 Functions

```
def area(l, w=1):        # w has a default value
    return l * w

def min_max(nums):
    return min(nums), max(nums)    # returns a tuple

lo, hi = min_max([4, 9, 1])        # unpack: lo=1, hi=9
square = lambda v: v * v           # small anonymous function
```

Inner functions can read outer variables. To **reassign** an outer variable inside a nested function, declare it `nonlocal`. This is common in DFS helpers that update a best answer.

```
def max_depth(root):
    best = 0
    def dfs(node, d):
        nonlocal best          # without this: UnboundLocalError
        if not node: return
        best = max(best, d)
        dfs(node.left, d + 1); dfs(node.right, d + 1)
    dfs(root, 1)
    return best
```

✓ CHECK YOUR PROGRESS 0.2

1. Write a range that gives 10, 8, 6, 4.

Answer: `range(10, 3, -2)` . The stop value is excluded, so any stop from 3 down to 2 works.

2. When do you need `nonlocal` ?

Answer: When a nested function assigns to a variable of the enclosing function. Reading works without it; assigning creates a new local unless you say `nonlocal` . Mutating a list (`res.append`) does not need it.

0.3 Lists and Strings

0.3.1 Lists: indexing, slicing, append and pop

A list is a dynamic array (like Java's `ArrayList`). Negative indexes count from the end.

```
a = [10, 20, 30, 40, 50]
a[0], a[-1]      # 10, 50
a[1:4]           # [20, 30, 40] start included, end excluded
a[:2], a[3:]     # [10, 20], [40, 50]
a[::-1]          # reversed copy
a[::-2]          # [10, 30, 50]
a.append(60)     # O(1) add at end
a.pop()          # O(1) remove last, returns 60
a.pop(0)         # O(n)! shifts everything. Use deque instead
a.insert(1, 15)  # O(n)
a.sort()         # in place, returns None
b = sorted(a)    # new sorted list
len(a), sum(a), max(a), min(a)
a.index(30)      # O(n) search, error if missing
```

A list works as a **stack** directly: `append` to push, `pop()` to pop, `stack[-1]` to peek.

0.3.2 List comprehensions

```
squares = [x * x for x in range(5)]      # [0,1,4,9,16]
evens   = [x for x in nums if x % 2 == 0] # filter
pairs   = [(i, j) for i in range(3) for j in range(i)] # nested
```

Comprehensions are shorter and usually a bit faster than a loop with `append`.

0.3.3 2D lists and the `[[0]*m]*n` trap

```
n, m = 3, 4
bad = [[0] * m] * n          # 3 references to the SAME row
bad[0][0] = 1
print(bad)                   # [[1,0,0,0],[1,0,0,0],[1,0,0,0]] !!
grid = [[0] * m for _ in range(n)] # correct: a new row each time
grid[0][0] = 1               # only row 0 changes
```

KEY POINT

`[0] * m` is safe because ints are immutable. `[[0] * m] * n` copies the reference to one list `n` times. Always build DP tables and grids with a comprehension.

0.3.4 Strings

Strings are **immutable**. `s[0] = "x"` raises an error. To build a string, collect pieces in a list and `"".join` them at the end. Repeated `s += ch` in a loop can become $O(n^2)$.

```
s = "chai"
s[::-1]                # "iahc"
s = s[::-1]            # palindrome check
ord("a"), chr(98)      # 97, "b"
idx = ord(ch) - ord("a") # letter to 0..25, used in count arrays
chars = list(s); chars[0] = "C"; s2 = "".join(chars) # "Chai"
" ".join(["a", "b"])   # "a b"
"a,b,c".split(",")     # ["a","b","c"]
" hi ".strip()         # "hi"
s.isdigit(), s.isalpha(), s.isalnum(), s.lower(), s.upper()
s.find("ai")           # 2, or -1 if missing
f"{s} costs {10}"      # f-string formatting
```

FUNNY SIDE

Candidate: "I made a 1000×1000 DP grid with `[[0]*m]*n`." Interviewer: "Why is every row the same?" Candidate: "Unity in diversity, sir." The Mumbai local can carry thousands of people, but this grid carries only one row, a thousand times.

✓ CHECK YOUR PROGRESS 0.3

1. What is `"python"[1:-1]` ?

Answer: `"ytho"`. Index 1 to the second last, end excluded.

2. Why is `"".join(parts)` preferred over `s += part` in a loop?

Answer: Strings are immutable, so each `+=` may copy the whole string, giving $O(n^2)$ in the worst case. `join` computes the total size once and copies each piece once: $O(n)$.

0.4 Tuples, Sets and Dicts

0.4.1 Tuples

A tuple is an immutable list: `p = (2, 3)`. Because it cannot change, it is **hashable**, so it can be a dict key or set item. Lists cannot. Use tuples for grid cells `(r, c)`, visited states, and memo keys.

```
visited = set()
visited.add((0, 1))          # OK
# visited.add([0, 1])        # TypeError: unhashable type: 'list'
single = (5,)                # one-element tuple needs the comma
```

0.4.2 Sets

```
seen = set()                 # NOT {} : that is an empty dict
seen.add(3); seen.discard(9) # discard: no error if missing
3 in seen                    # O(1) average
a, b = {1, 2, 3}, {2, 3, 4}
a & b, a | b, a - b           # {2,3}, {1,2,3,4}, {1}
```

0.4.3 Dicts

A dict is a hash map. Since Python 3.7, it keeps insertion order.

```
age = {"asha": 25}
age["ravi"] = 30
age.get("meena", 0)          # 0 instead of KeyError
age.setdefault("om", []).append(1) # create if missing, then use
for k, v in age.items(): print(k, v)
"asha" in age                 # O(1) key check
del age["ravi"]
freq = {}
for ch in "banana":
    freq[ch] = freq.get(ch, 0) + 1 # {'b':1, 'a':3, 'n':2}
```

INTERVIEW CORNER

Q: Why can a tuple be a dict key but a list cannot?

A: Keys must be hashable, and the hash must never change. A list can be mutated, which would change its hash and lose it inside the table. A tuple of immutable items is fixed, so its hash is stable.

Q: Is `x in my_list` the same speed as `x in my_set`?

A: No. A list scans every element: $O(n)$. A set hashes: $O(1)$ on average. Inside a loop, this is the difference between $O(n^2)$ and $O(n)$.

✓ CHECK YOUR PROGRESS 0.4

1. What does `type({})` return?

Answer: `dict`. An empty set must be written `set()`.

2. What is the difference between `d[k]` and `d.get(k)`?

Answer: `d[k]` raises `KeyError` if `k` is missing. `d.get(k)` returns `None` (or a default you pass) instead.

0.5 The collections Module

```
from collections import Counter, defaultdict, deque, OrderedDict
```

0.5.1 Counter

```
c = Counter("banana")          # Counter({'a':3, 'n':2, 'b':1})
c["z"]                         # 0, no KeyError
c.most_common(2)               # [('a',3), ('n',2)]
Counter("listen") == Counter("silent") # True: anagram check
```

0.5.2 defaultdict

```
graph = defaultdict(list)      # missing key starts as []
for u, v in [(1, 2), (1, 3)]:
    graph[u].append(v)
    graph[v].append(u)
groups = defaultdict(list)
for w in ["eat", "tea", "tan"]:
    groups["".join(sorted(w))].append(w) # group anagrams
```

0.5.3 deque

A double-ended queue: $O(1)$ add and remove at both ends. Use it for BFS queues and sliding windows. Never use `list.pop(0)` as a queue.

```
q = deque([1])
q.append(2); q.appendleft(0) # [0,1,2]
q.popleft(); q.pop()        # O(1) both
last3 = deque(maxlen=3)     # auto-drops oldest item when full
```

0.5.4 OrderedDict

Normal dicts keep order now, but `OrderedDict` adds `move_to_end(key)` and `popitem(last=False)` in $O(1)$. That is exactly what an LRU cache needs.

```
cache = OrderedDict()
cache["a"] = 1; cache["b"] = 2
cache.move_to_end("a")      # "a" is now most recent
cache.popitem(last=False)   # removes "b", the least recent
```

AT WORK

A backend developer at a food delivery start-up wrote a log parser that counted error codes with a manual dict and if-else. In code review, it became one line: `Counter(code for code in codes).most_common(5)`. The same `deque(maxlen=N)` trick is used to keep "the last N requests" for rate limiting or debugging dashboards.

✓ CHECK YOUR PROGRESS 0.5

1. Which structure would you use for a BFS queue, and why not a list?

Answer: `deque`, because `popleft()` is $O(1)$. `list.pop(0)` shifts every element, which is $O(n)$ per pop.

2. What does `defaultdict(int)` give for a missing key?

Answer: 0, since `int()` returns 0. The key is also inserted into the dict.

0.6 heapq and bisect

0.6.1 heapq: a min-heap on a plain list

```
import heapq
h = []
heapq.heappush(h, 5); heapq.heappush(h, 1); heapq.heappush(h, 3)
h[0]                # 1, the smallest (peek)
heapq.heappop(h)     # 1,  $O(\log n)$ 
nums = [4, 1, 7]
heapq.heapify(nums)  #  $O(n)$ , in place
heapq.heappushpop(h, 2) # push 2, then pop smallest (fast combo)
heapq.nlargest(2, [4, 1, 7, 3]) # [7, 4]
heapq.nsmallest(2, [4, 1, 7, 3]) # [1, 3]
```

0.6.2 Max-heap trick and tuples

Python has only a min-heap. For a max-heap, push negatives and negate when popping. Tuples compare item by item, so `(priority, item)` works naturally.

```
mx = []
for x in [4, 9, 2]:
    heapq.heappush(mx, -x)
-mx[0]                # 9, the largest
heapq.heappush(pq, (dist, node)) # sorted by dist first
```

⚠ COMMON MISTAKE

If two tuples have equal priority, Python compares the next item. If that item is a `ListNode` or other object without `<`, you get a `TypeError`. Fix: push `(priority, counter, obj)` or `(val, i, node)` with a unique index in the middle.

0.6.3 bisect: binary search on a sorted list

```
import bisect
a = [1, 3, 3, 5]
bisect.bisect_left(a, 3)      # 1: first index where a[i] ≥ 3
bisect.bisect_right(a, 3)     # 3: first index where a[i] > 3
bisect.bisect_left(a, 4)      # 3: insertion point
bisect.insort(a, 4)           # [1,3,3,4,5], O(n) due to shift
count3 = bisect.bisect_right(a, 3) - bisect.bisect_left(a, 3)  # 2
```

✓ CHECK YOUR PROGRESS 0.6

1. How do you get the 3 largest numbers of a list with `heapq`?

Answer: `heapq.nlargest(3, nums)`, about $O(n \log 3)$. Or keep a min-heap of size 3 and pop when it grows past 3.

2. In `[2, 4, 4, 8]`, what are `bisect_left(a, 4)` and `bisect_right(a, 4)`?

Answer: 1 and 3.

0.7 Sorting, enumerate and zip**0.7.1 Sorting with key and lambda**

```
words = ["kiwi", "fig", "banana"]
sorted(words, key=len)           # ['fig','kiwi','banana']
sorted(words, key=len, reverse=True)
people = [("asha", 25), ("ravi", 22), ("om", 25)]
people.sort(key=lambda p: (-p[1], p[0])) # age desc, then name asc
intervals.sort(key=lambda iv: iv[0])    # by start time
```

Python's sort is **stable**: equal items keep their original order. The tuple key with a minus sign is the standard trick for "descending on one field, ascending on another".

0.7.2 cmp_to_key for custom comparisons

When the order cannot be expressed as a key, write a Java-style comparator (negative, zero, positive) and wrap it.

```

from functools import cmp_to_key
def largest_number(nums):          # [3,30,34] → "34330"
    strs = list(map(str, nums))
    def cmp(a, b):
        if a + b > b + a: return -1    # a should come first
        if a + b < b + a: return 1
        return 0
    strs.sort(key=cmp_to_key(cmp))
    res = "".join(strs)
    return "0" if res[0] == "0" else res    # case [0,0]

```

0.7.3 enumerate and zip

```

for i, x in enumerate(["a", "b"]):    # 0 a, 1 b
    print(i, x)
for i, x in enumerate(nums, start=1): # index starts at 1
    pass
for name, score in zip(names, scores): # stops at the shorter list
    print(name, score)
transposed = [list(row) for row in zip(*matrix)]
diffs = [b - a for a, b in zip(nums, nums[1:])]    # adjacent pairs

```

💡 DO YOU KNOW?

Python's sorting algorithm, **Timsort**, was written by Tim Peters for Python in 2002. It is a hybrid of merge sort and insertion sort that is very fast on partly sorted data. Java adopted Timsort for sorting objects in Java 7, and it is also used in Android and V8 (Chrome's JavaScript engine).

✅ CHECK YOUR PROGRESS 0.7

1. Sort a list of (name, marks) by marks descending, then name ascending.

Answer: `lst.sort(key=lambda t: (-t[1], t[0]))`.

2. What does `list(zip([1, 2, 3], "ab"))` return?

Answer: `[(1, 'a'), (2, 'b')]`. `zip` stops at the shorter input.

0.8 Recursion, Memoization and Handy Modules

0.8.1 Recursion and the depth limit

A recursive function calls itself on a smaller input and stops at a base case. Python's default recursion limit is about 1000 frames. A DFS on a linked list or path graph of 10^5 nodes will crash with `RecursionError`.

```
import sys
sys.setrecursionlimit(10**6)  # raise the limit

def fact(n):
    if n ≤ 1:                  # base case
        return 1
    return n * fact(n - 1)
```

⚠ COMMON MISTAKE

Raising the limit is not a full fix. Very deep recursion can still overflow the real C stack and crash the process silently. For inputs of 10^5 or more depth, convert DFS to an iterative version with an explicit stack.

0.8.2 lru_cache and cache

Memoization stores answers of subproblems so each is computed once. Python does it with one decorator. Arguments must be hashable (use tuples, not lists).

```
from functools import lru_cache, cache  # cache: Python 3.9+

@lru_cache(maxsize=None)
def fib(n):
    if n < 2:
        return n
    return fib(n - 1) + fib(n - 2)  # O(n) instead of O(2^n)

fib(80)  # 23416728348467685, instant
fib.cache_clear()  # reset between test cases if needed
```

Without memo, `fib(40)` makes over 300 million calls. With memo, about 40 distinct calls do the real work.

0.8.3 Infinity and the math module

```
import math
best = float("inf")  # larger than any number
worst = float("-inf")
math.inf  # same as float("inf")
math.gcd(12, 18)  # 6
math.lcm(4, 6)  # 12 (Python 3.9+)
math.isqrt(17)  # 4, exact integer square root
math.ceil(7 / 2), math.floor(-3.5)  # 4, -4
math.comb(5, 2)  # 10 ways to choose 2 of 5
pow(2, 10, 1000)  # 24: fast modular power, built-in
abs(-5), divmod(17, 5)  # 5, (3, 2)
```

🔑 KEY POINT

For ceiling division of positive ints, use `(a + b - 1) // b` or `-(-a // b)`. It avoids float error that `math.ceil(a / b)` can give for very large numbers.

✓ CHECK YOUR PROGRESS 0.8

1. Your memoized function takes a list argument and crashes. Why?

Answer: `lru_cache` stores arguments as dict keys, so they must be hashable. Pass a tuple instead of a list.

2. Why is `math.isqrt(n)` better than `int(math.sqrt(n))`?

Answer: `math.sqrt` uses floats and can be off by one for very large `n`. `isqrt` is exact for any int.

0.9 Fast Input and the Python Complexity Table

0.9.1 Fast input with `sys.stdin`

LeetCode passes arguments to your function, so input speed does not matter there. On HackerRank, Codeforces or company test platforms you read from standard input. `input()` is slow for 10^5 + lines.

```
import sys
input = sys.stdin.readline      # much faster
n = int(input())
arr = list(map(int, input().split()))
s = input().strip()            # strip the trailing newline!

data = sys.stdin.buffer.read().split() # read everything at once
n = int(data[0]); nums = list(map(int, data[1:1 + n]))
sys.stdout.write("\n".join(map(str, results)) + "\n") # fast output
```

0.9.2 Complexity of common operations

Operation	list	dict / set	deque
Append / add at end	$O(1)$ amortized	$O(1)$ avg	$O(1)$
Pop from end	$O(1)$	n/a	$O(1)$
Insert / pop at front	$O(n)$	n/a	$O(1)$
Access by index	$O(1)$	$O(1)$ by key	$O(n)$ in middle, $O(1)$ at ends
<code>x in ...</code>	$O(n)$	$O(1)$ avg	$O(n)$
Delete by value / key	$O(n)$	$O(1)$ avg	$O(n)$
Slice of length <code>k</code>	$O(k)$	n/a	n/a
Sort	$O(n \log n)$	n/a	n/a
min / max / sum	$O(n)$	$O(n)$	$O(n)$
len	$O(1)$	$O(1)$	$O(1)$

Table 0.2: Big-O of Python built-ins (heap push/pop $O(\log n)$, heapify $O(n)$, bisect search $O(\log n)$, string slice and copy $O(k)$)

A rough rule: Python does about 10^7 simple operations per second, slower than Java. With $n = 10^5$, $O(n \log n)$ is fine; $O(n^2)$ (10^{10}) will time out.

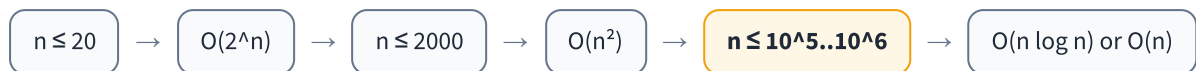


Figure 0.1: guessing the needed complexity from input size

✓ CHECK YOUR PROGRESS 0.9

1. You do `if x in arr` inside a loop over `arr`, with $n = 10^5$. What happens and how do you fix it?

Answer: It is $O(n^2) = 10^{10}$ steps and will time out. Convert to a set once (`s = set(arr)`) and check `x in s` for $O(n)$ total.

2. Why add `.strip()` after `sys.stdin.readline()` for strings?

Answer: `readline` keeps the trailing `"\n"`, which would break comparisons and length checks. `int()` ignores it, but strings do not.

0.10 Common Python Mistakes

0.10.1 Mutable default arguments

```
def add(x, bucket=[]):      # BUG: one list shared by all calls
    bucket.append(x); return bucket
add(1); add(2)              # second call returns [1, 2]!

def add(x, bucket=None):    # FIX
    if bucket is None:
        bucket = []
    bucket.append(x); return bucket
```

0.10.2 Copying lists

```
a = [1, 2, 3]
b = a                      # same list, not a copy
c = a[:]                   # shallow copy (also a.copy(), list(a))
import copy
g2 = copy.deepcopy(grid)   # needed for nested lists
res.append(path[:])        # backtracking: save a COPY of path
```

In backtracking, `res.append(path)` stores a reference. After you pop elements, every saved answer becomes empty. Always append `path[:]`.

0.10.3 Modifying a list while iterating

```
nums = [2, 4, 5, 6]
for x in nums:
    if x % 2 == 0:
        nums.remove(x)      # BUG: skips elements, result [4, 5]
nums = [x for x in nums if x % 2]  # FIX: build a new list
for k in list(d.keys()):      # dicts: iterate over a copy of keys
    if d[k] == 0: del d[k]
```

0.10.4 Other quick traps

- `is` checks identity, `=` checks value. Use `is` only for `None`.
- `list.sort()` returns `None`. `a = a.sort()` loses your list.
- Shadowing built-ins: naming a variable `list`, `sum` or `max` breaks them later.
- `True == 1` is true; a `bool` is a kind of `int`, so `sum(flags)` counts `True` values.

INTERVIEW CORNER

Q: Your backtracking returns a list of empty lists. What went wrong?

A: I appended the same `path` object, then popped from it. I should append `path[:]` so each answer is a snapshot.

Q: Why use Python for interviews if it is slower than Java?

A: Interview limits are about algorithm complexity, not constant factors. Python lets me write the correct algorithm in fewer lines, so I spend more time on the idea and edge cases. If a judge is tight on time, I use fast input and avoid deep recursion.

ACTIVITY 0.1

Open a Python shell and run: `g = [[0]*3]*3; g[1][1] = 9; print(g)`, then `print(-7 // 2, -7 % 3, 0.1 + 0.2)`. Predict each output on paper first. Every wrong guess is a bug you will not make in the real round.

THINK AND DISCUSS

Python hides many details (no overflow, auto-resizing lists, built-in heaps). Is it fair to use `Counter` or `heapq.nlargest` in an interview, or should you implement them? When might an interviewer ask you to build them yourself?

✓ CHECK YOUR PROGRESS 0.10

1. Why does `def f(x, seen=set())` cause bugs across test cases?

Answer: The default set is created once when the function is defined and shared by every call. Items from earlier calls remain. Use `None` as default and create the set inside.

2. Is `a[:]` enough to copy a 2D grid?

Answer: No. It copies the outer list, but the rows are still shared. Use `[row[:] for row in grid]` or `copy.deepcopy`.

0.11 Fifteen Warm-up Problems

Solve each yourself first (5 to 10 minutes), then compare. Each solution is short because Python does the heavy lifting.

0.11.1 Sum of digits (Easy)

```
def digit_sum(n):          # 1234 → 10
    n, total = abs(n), 0
    while n:
        total += n % 10; n //= 10
    return total
```

0.11.2 Valid palindrome, letters and digits only (Easy)

```
def is_pal(s):             # "A man, a plan..." → True
    t = [c.lower() for c in s if c.isalnum()]
    return t == t[::-1]
```

0.11.3 Count vowels (Easy)

```
def vowels(s):
    return sum(1 for c in s.lower() if c in "aeiou")
```

0.11.4 Maximum without max() (Easy)

```
def my_max(nums):
    best = float("-inf")
    for x in nums:
        if x > best: best = x
    return best
```

0.11.5 Second largest distinct value (Easy)

```
def second_largest(nums):    # [5,5,3] → 3, [7] → None
    first = second = float("-inf")
    for x in nums:
        if x > first: first, second = x, first
        elif first > x > second: second = x
    return None if second == float("-inf") else second
```

0.11.6 First non-repeating character (Easy)

```
from collections import Counter
def first_unique(s):        # "leetcode" → 0
    c = Counter(s)
    for i, ch in enumerate(s):
        if c[ch] == 1: return i
    return -1
```

0.11.7 Two Sum (Easy, LeetCode 1)

```
def two_sum(nums, target):  # [2,7,11,15], 9 → [0,1]
    pos = {}
    for i, x in enumerate(nums):
        if target - x in pos:      # complement seen before?
            return [pos[target - x], i]
        pos[x] = i
    return []
```

Complexity: $O(n)$ time, $O(n)$ space. Brute force with two loops is $O(n^2)$.

0.11.8 Remove duplicates, keep order (Easy)

```
def dedup(nums):            # [3,1,3,2,1] → [3,1,2]
    return list(dict.fromkeys(nums))  # dict keeps insertion order
```

0.11.9 Valid anagram (Easy, LeetCode 242)

```
def is_anagram(s, t):
    return Counter(s) == Counter(t)    #  $O(n)$ ; sorted(s)==sorted(t) is  $O(n \log n)$ 
```

0.11.10 FizzBuzz (Easy)

```
def fizzbuzz(n):
    out = []
    for i in range(1, n + 1):
        s = ("Fizz" if i % 3 == 0 else "") + ("Buzz" if i % 5 == 0 else "")
        out.append(s or str(i))        # empty string is falsy
    return out
```

0.11.11 Climbing stairs with memo (Easy, LeetCode 70)

```
from functools import cache
def climb(n):
    # ways to climb n steps by 1 or 2
    @cache
    def f(i):
        if i ≤ 1: return 1
        return f(i - 1) + f(i - 2)
    return f(n)
```

0.11.12 GCD and LCM of a list (Easy)

```
import math
from functools import reduce
def gcd_list(nums): return reduce(math.gcd, nums)
def lcm_list(nums): return reduce(lambda a, b: a * b // math.gcd(a, b), nums)
```

0.11.13 Transpose a matrix (Easy, LeetCode 867)

```
def transpose(m):
    # [[1,2,3],[4,5,6]] → [[1,4],[2,5],[3,6]]
    return [list(col) for col in zip(*m)]
```

0.11.14 Running sum / prefix sums (Easy, LeetCode 1480)

```
from itertools import accumulate
def running_sum(nums):
    # [1,2,3,4] → [1,3,6,10]
    return list(accumulate(nums))
# sum of nums[l..r] = pre[r+1] - pre[l] where pre = [0] + running_sum(nums)
```

0.11.15 Top K frequent elements (Medium, LeetCode 347)

```
import heapq
def top_k(nums, k):
    # [1,1,1,2,2,3], 2 → [1,2]
    c = Counter(nums)
    return heapq.nlargest(k, c.keys(), key=c.get) # O(n log k)
# or: [x for x, _ in c.most_common(k)]
```

✓ CHECK YOUR PROGRESS 0.11**1. In Two Sum, why check the complement before storing the current number?**

Answer: So a number does not pair with itself. For [3, 4], target 6, storing 3 first and then checking would wrongly return [0, 0].

2. Why does `dict.fromkeys(nums)` remove duplicates while keeping order?

Answer: Dict keys are unique, and dicts preserve insertion order (Python 3.7+), so the first occurrence of each value stays in place.

What Have We Learnt?

- Python ints never overflow, but you still apply `% MOD` when asked.
- `//` and `%` floor towards minus infinity; use `int(a / b)` for truncation.
- Never compare floats with `=`; prefer integer arithmetic.
- Lists are dynamic arrays: fast at the end, slow at the front.
- Build grids with `[[0] * m for _ in range(n)]`, never `[[0]*m]*n`.
- Strings are immutable; build with a list and `"".join`.
- Tuples are hashable, so they work as dict keys and set items.
- `Counter`, `defaultdict`, `deque` and `OrderedDict` cover most counting, grouping, queue and LRU needs.
- `heapq` is a min-heap; negate values for a max-heap and add a tie-breaker index for objects.
- `bisect_left` and `bisect_right` give binary search on sorted lists.
- Sort with `key=lambda tuples`; use `cmp_to_key` for pairwise rules.
- Raise the recursion limit or go iterative for deep recursion; memoize with `@cache`.
- Use `sys.stdin.readline` for fast input on stdin-based judges.
- Watch for mutable defaults, shallow copies and modifying a list while looping.

Key terms: arbitrary precision, floor division, falsy, slicing, list comprehension, immutability, hashable, Counter, defaultdict, deque, OrderedDict, heapq, bisect, stable sort, cmp_to_key, memoization, lru_cache, recursion limit, shallow copy, deep copy, amortized O(1)

Exercises

A. Multiple Choice Questions

- What is `-11 // 3` in Python?
 - `-3`
 - `-4`
 - `-3.67`
 - `3`
- Which is the correct way to create a 3×3 grid of zeros?
 - `[[0]*3]*3`
 - `[0]*9`
 - `[[0]*3 for _ in range(3)]`
 - `[[0]]*3*3`
- Time complexity of `deque.popleft()` ?
 - $O(n)$
 - $O(\log n)$
 - $O(1)$
 - $O(n \log n)$
- Which can be stored in a set?
 - `[1, 2]`
 - `(1, 2)`
 - `{1: 2}`
 - `{1, 2}`
- How do you peek at the maximum of a max-heap `h` built with negatives?
 - `h[0]`
 - `-h[0]`
 - `h[-1]`
 - `max(h)`

B. Fill in the Blanks

- To read input fast, assign `input = _____`.
- `bisect._____ (a, x)` returns the first index where `a[i] > x`.
- The default recursion limit in CPython is about _____.

C. True or False

- `list.sort()` returns the sorted list.
- Python's built-in sort is stable.
- `x in some_dict` checks values, not keys.

D. Short Answer Questions (practice problems)

- Contains Duplicate:** return True if any value appears twice. Hint: compare lengths.
- Reverse Words in a String:** "the sky is blue" → "blue is sky the". Hint: split with no argument.
- Kth Largest Element in an Array:** Hint: min-heap of size k.

E. Long Answer Questions

- Explain list, set, dict, deque and heap in Python: what each is, its main operations with Big-O, and one interview use of each.

2. **Group Anagrams:** group words that are anagrams of each other. Explain two possible keys and their complexity, and write the code.

F. Design Challenge

Build a simple **LRU cache** class with `get(key)` and `put(key, value)`, both $O(1)$, and a fixed capacity. Use only Python built-ins.

Answer Key: Chapter 0

A. MCQ

1. (b) Floor of -3.67 is -4 .
2. (c) Each row is a new list.
3. (c) deque is $O(1)$ at both ends.
4. (b) Only tuples are hashable among these.
5. (b) Stored values are negated, so negate back.

B. Fill in the Blanks

1. `sys.stdin.readline`
2. `bisect_right`
3. 1000

C. True or False

1. False. It sorts in place and returns None; `sorted()` returns a new list.
2. True. Timsort keeps equal items in original order.
3. False. It checks keys in $O(1)$ average; checking values needs `x in d.values()`, which is $O(n)$.

D. Short Answers

1. `return len(set(nums)) != len(nums)`. A set drops duplicates, so a smaller size means a repeat. $O(n)$ time and space.
2. `return " ".join(s.split()[::-1])`. `split()` with no argument handles multiple spaces and trims ends. $O(n)$.
3. Push each number into a min-heap; when size exceeds k , pop. The top `h[0]` is the answer. $O(n \log k)$ time, $O(k)$ space. One-liner: `heapq.nlargest(k, nums)[-1]`.

E. Long Answers

1. Model answer:
 - list: dynamic array; index $O(1)$, append/pop end $O(1)$ amortized, front insert $O(n)$, search $O(n)$. Use as a stack (valid parentheses).
 - set: hash set; add/remove/in $O(1)$ average. Use for visited nodes or duplicate checks.
 - dict: hash map; get/set/in $O(1)$ average, keeps insertion order. Use for Two Sum complements or frequency maps.
 - deque: double-ended queue; $O(1)$ at both ends, $O(n)$ in the middle. Use for BFS and sliding window maximum.
 - heapq: binary min-heap on a list; push/pop $O(\log n)$, peek $O(1)$, heapify $O(n)$. Use for top-k, Dijkstra, merge k sorted lists.
 - Wrong choice turns $O(n)$ into $O(n^2)$, for example `in list` or `pop(0)` inside a loop.
2. Model answer:
 - Key 1: `" ".join(sorted(w))`, cost $O(L \log L)$ per word.
 - Key 2: a 26-count tuple, cost $O(L)$ per word: `tuple` because lists are unhashable.

- Use `defaultdict(list)` to collect groups.
- Total $O(n \cdot L \log L)$ or $O(n \cdot L)$; space $O(n \cdot L)$.

```
def group_anagrams(words):
    groups = defaultdict(list)
    for w in words:
        cnt = [0] * 26
        for ch in w:
            cnt[ord(ch) - ord("a")] += 1
        groups[tuple(cnt)].append(w)  # tuple is hashable
    return list(groups.values())
```

F. Design Challenge: Model Solution

Use `OrderedDict`: it keeps keys in order of use, and `move_to_end` and `popitem(last=False)` are $O(1)$. On get, move the key to the end (most recent). On put, insert or update, move to end, and if size exceeds capacity, pop the first (least recent) item.

```
from collections import OrderedDict
class LRUCache:
    def __init__(self, capacity):
        self.cap = capacity
        self.data = OrderedDict()

    def get(self, key):
        if key not in self.data:
            return -1
        self.data.move_to_end(key)  # mark as recently used
        return self.data[key]

    def put(self, key, value):
        self.data[key] = value
        self.data.move_to_end(key)
        if len(self.data) > self.cap:
            self.data.popitem(last=False)  # evict least recent
```

In an interview, mention that the "from scratch" version uses a dict plus a doubly linked list, which is exactly what `OrderedDict` does internally.

CHAPTER 1

How to Approach Any DSA Problem

"The interviewer is not checking if you have seen the problem. They are checking if you can think when you have not."

After studying this chapter, you will be able to

- Follow a seven-step framework for any coding interview problem.
- Read input constraints and guess the time complexity the interviewer expects.
- Map common problem signals to the right pattern and chapter of this book.
- Dry run code on paper using a step table, and catch bugs before the interviewer does.
- Talk clearly while you code, so the interviewer can follow and help you.
- Use Python lists, dicts, sets, deque, heapq and sorting keys fluently, and avoid the common Python traps.

Think of a chai stall owner at a busy railway station. Twenty people shout orders at once: "two cutting, one without sugar, one masala, one ginger". A new helper panics and starts making cups one by one in the order he heard them. The owner does something different. He first listens, groups the orders (four cutting, two masala), checks how much milk he has, then makes everything in one go. Same job, very different result.

Coding interviews are like that. Most candidates who fail do not fail because they do not know arrays or hash maps. They fail because they jump into code without a plan, get stuck in the middle, and go silent. This chapter gives you the owner's method: a fixed set of steps, a way to read the constraints like a menu card, a pattern map for the rest of the book, and a quick refresher of the Python tools you will use every single day of your preparation.

1.1 The Interview Problem-Solving Framework

Every problem in this book, easy or hard, can be solved with the same seven steps. Use them until they become habit.



Figure 1.1: The seven-step framework

1.1.1 Step 1: Understand the problem

Read the problem twice. Then say it back in your own words: "So I get an array of integers, and I must return the indices of two numbers that add up to target. Exactly one answer exists." Now ask clarifying questions. Good questions are short and change the solution:

- Can the input be empty? Can it have one element?
- Are there negative numbers? Zeros? Duplicates?
- Is the array sorted? Can I modify it?
- What should I return if there is no answer: -1, empty list, or throw?
- How large can n be? How large can the values be (could floats or huge numbers appear)?

1.1.2 Step 2: Work through examples

Take the given example and solve it by hand. Then make one small example of your own and one edge case. Hand-solving often shows you the algorithm. If you cannot solve it by hand, you cannot code it.

1.1.3 Step 3: State the brute force

Say the simplest correct solution out loud, with its complexity, even if it is slow. "I can try every pair, that is $O(n^2)$ time and $O(1)$ space." This proves you understand the problem and gives you a safe fallback. Do not code it unless the interviewer asks.

1.1.4 Step 4: Optimise

Look at the brute force and ask: what work is repeated? Common tools that remove repeated work:

- **Hashing:** replace "search for x " ($O(n)$) with a dict or set lookup ($O(1)$ average).
- **Sorting:** once sorted, two pointers or binary search often apply.
- **Precomputation:** prefix sums, prefix max, frequency counts.
- **Sliding window:** reuse the previous window instead of recomputing.
- **Better data structure:** heap for "top k ", stack for "next greater", deque for "window max".
- **Caching recursion:** memoisation turns exponential recursion into DP.

Use the BUD idea: look for **B**ottlenecks, **U**nnecessary work and **D**uplicated work.

1.1.5 Step 5: Code

Only now write code. Write clean Python: good names (`left` , `right` , `seen`), small helper functions, and no clever one-liners. If a part is boring (like parsing), say "I will assume a helper here" and move on.

1.1.6 Step 6: Test

Do not say "I think it works". Dry run your code (not your idea) on the small example, then on edge cases: empty input, one element, all same values, negatives, very large values. Section 1.4 shows how.

1.1.7 Step 7: Analyse

State time and space complexity, and why. Mention the trade-off: "I use $O(n)$ extra space to bring time from $O(n^2)$ to $O(n)$." Then offer a follow-up if there is time.

1.1.8 The framework applied: Two Sum

Problem: given `nums = [2, 7, 11, 15]` and `target = 9`, return indices of the two numbers that add up to target (LeetCode 1).

Step	What you say or do
Understand	"Exactly one answer, I cannot use the same element twice, return indices."
Examples	[2,7,11,15], 9 → [0,1]. Own case: [3,3], 6 → [0,1] (duplicates).
Brute force	Check every pair $i < j$. $O(n^2)$ time, $O(1)$ space.
Optimise	For each x , I am searching for $target - x$. A dict of value → index makes the search $O(1)$.
Code	See below.
Test	[3,3]: $i=0$ stores $3 \rightarrow 0$; $i=1$ finds 3 in map, returns [0,1]. Correct.
Analyse	$O(n)$ time, $O(n)$ space.

```
class Solution:
    def twoSum(self, nums: list[int], target: int) → list[int]:
        index_of: dict[int, int] = {}
        for i, x in enumerate(nums):
            need = target - x
            if need in index_of:                # partner seen earlier
                return [index_of[need], i]
            index_of[x] = i                    # store AFTER the check
        return [-1, -1]                       # no pair found
```

KEY POINT

Brute force first, out loud, with its complexity. It is never wasted time. It shows understanding, and the optimised solution almost always comes from staring at what the brute force repeats.

COMMON MISTAKE

Coding before agreeing on the approach. If you code a wrong idea for 15 minutes, there is no time left to fix it. Always say "Shall I code this approach?" and wait for a nod.

✓ CHECK YOUR PROGRESS 1.1**1. Why should you state the brute-force solution even when you know a better one?**

Answer: It proves you understand the problem, gives a correct fallback, and gives a baseline complexity to improve on. Interviewers often give partial credit for a clear brute force.

2. Name three clarifying questions that can change an array solution.

Answer: Is the array sorted? Can it contain negatives or duplicates? Can I modify the input? (Also: size of n, empty input behaviour.)

1.2 Reading Constraints to Guess the Target Complexity

Online judges and most interviewers give constraints like $1 \leq n \leq 10^5$. These numbers are a hint. A typical judge runs roughly 10^8 simple operations per second (a rough rule of thumb, not an exact number). So the allowed complexity is whatever keeps the operation count near or below about 10^8 .

1.2.1 The constraint table

Input size n	Target complexity	Typical technique
$n \leq 10$	$O(n!)$ or $O(n \cdot n!)$	All permutations, backtracking
$n \leq 20$	$O(2^n)$ or $O(n \cdot 2^n)$	Subsets, bitmask, backtracking
$n \leq 500$	$O(n^3)$	Triple loops, interval DP, Floyd-Warshall
$n \leq 5,000$	$O(n^2)$	Double loops, 2D DP (LCS, edit distance)
$n \leq 10^6$	$O(n \log n)$ or $O(n)$	Sorting, heap, two pointers, hashing, sliding window
$n \geq 10^9$	$O(\log n)$ or $O(1)$	Binary search, math, fast exponentiation

1.2.2 Worked examples

- $n \leq 10^5$:** $n^2 = 10^{10}$ operations, far too slow. $n \log n \approx 10^5 \times 17 \approx 1.7 \times 10^6$, very fast. So aim for $O(n \log n)$ or $O(n)$.
- $n \leq 20$:** $2^{20} \approx 10^6$. Trying every subset is fine. The small limit is the interviewer saying "exponential is okay".
- $n \leq 10^9$** (for example, "find the square root of n"): even $O(n)$ is 10^9 steps, too slow. Think binary search: $\log_2(10^9) \approx 30$ steps.
- Two inputs, $m, n \leq 1000$:** $O(m \cdot n) = 10^6$, fine. This hints at a 2D DP grid.

🔑 KEY POINT

Small n is a hint, not a gift. When $n \leq 20$, the problem probably wants exponential search. When n is huge, it wants logarithmic thinking. Read the constraints before you design.

1.2.3 Value constraints matter too

Look at the range of the values, not just n . Python `int` never overflows, but if values are up to 10^9 and you multiply several of them, the numbers grow large and arithmetic slows down; mixing in floats (like `/` or `math.sqrt`) can also lose precision above about 2^{53} . If values are small (0 to 100), a counting list of size 101 can replace a dict or sorting.

ACTIVITY 1.1

Open any five LeetCode problems you have solved. For each, write n 's limit and the complexity of your accepted solution. Check that every pair fits the table above. Find one where you could have guessed the approach from the constraint alone.

CHECK YOUR PROGRESS 1.2

1. A problem says $n \leq 2 \times 10^5$. Is an $O(n^2)$ solution acceptable?

Answer: No. $n^2 = 4 \times 10^{10}$, far above the roughly 10^8 operations per second budget. Aim for $O(n \log n)$ or $O(n)$.

2. What does $n \leq 15$ suggest?

Answer: Exponential is allowed: $2^{15} = 32,768$. Think subsets, bitmask DP or backtracking.

1.3 Pattern Recognition: The Map of This Book

There are thousands of problems but only about 15 to 20 core patterns. Interviews are mostly pattern matching: you read the problem, notice a signal word, and pick the right tool. The table below is your index. Come back to it often.

Problem signal	Likely pattern	Chapter
"Subarray sum", "range sum", many sum queries	Prefix sum (+ dict)	3
Sorted array, "pair with sum", "in place", palindrome	Two pointers	4
"Longest/shortest substring or subarray with condition"	Sliding window	5
"Count frequency", "seen before", "group by"	Hashing	6
Sorted input, "minimum value that works", huge n	Binary search (on answer)	7
"Next greater", "valid brackets", "previous smaller"	Stack / monotonic stack	11
"Reverse list", "cycle", "middle node"	Linked list, fast and slow pointers	10
Tree, "depth", "path", "level by level"	DFS / BFS on trees	12
"Top k", "k-th largest", "merge k sorted"	Heap (heapq)	13
"All combinations / permutations / subsets"	Backtracking	9
Grid, "islands", "shortest path", dependencies	Graph BFS / DFS / topological sort	14, 15

Problem signal	Likely pattern	Chapter
"Number of ways", "min cost", "max profit" with choices	Dynamic programming	16, 17
"Intervals", "meetings", "overlap"	Sort + greedy / merge intervals	18
"Connected groups", "redundant connection"	Union-Find	15

Table 1.1: Signal to pattern map (the last column is the chapter of this book)

1.3.1 How to use the map

1. Underline the nouns and verbs in the problem: "contiguous", "sorted", "k", "minimum", "all".
2. Check the output type: a number (count, min, max), a boolean, or a list of all solutions. "All solutions" almost always means backtracking.
3. Check the constraint table from Section 1.2 to confirm the pattern's complexity fits.
4. If two patterns fit, start with the simpler one.

FUNNY SIDE

Every DSA learner goes through three stages. Stage 1: "Every problem is new and scary." Stage 2: "Every problem is a dict." Stage 3: "Oh, this is just sliding window wearing a moustache." This book is the shortcut to Stage 3.

DO YOU KNOW?

LeetCode has more than 3,000 problems, yet popular curated lists such as "Blind 75" (posted by an engineer at Facebook around 2018) and "NeetCode 150" cover the core interview patterns with about 75 to 150 problems. Breadth of patterns beats raw count.

CHECK YOUR PROGRESS 1.3

1. A problem asks for "the length of the longest substring without repeating characters". Which pattern?

Answer: Sliding window with a set, or a dict of last-seen positions. The signal is "longest substring with a condition".

2. A problem asks for "all possible subsets". Which pattern, and what complexity should you expect?

Answer: Backtracking. There are 2^n subsets, so $O(n \cdot 2^n)$ is unavoidable; n will usually be ≤ 20 .

1.4 How to Dry Run

A dry run means executing your code by hand, line by line, tracking every variable in a table. It is the single best bug-catching habit. The interpreter catches syntax errors; dry runs catch logic.

1.4.1 The method

1. Pick a small input (3 to 6 elements) that touches the interesting cases.
2. Draw a table: one column per variable, one row per loop iteration.
3. Follow the **code you wrote**, not the idea in your head.
4. Check loop boundaries: does the first iteration and the last iteration do the right thing?
5. Repeat with an edge case: empty, one element, all equal.

1.4.2 Example: find the maximum

```
def find_max(a: list[int]) → int:
    best = 0                # suspicious start value
    for i in range(len(a)):
        if a[i] > best:
            best = a[i]
    return best
```

Dry run on `a = [-5, -2, -9]`:

i	a[i]	a[i] > best?	best
start	-	-	0
0	-5	no	0
1	-2	no	0
2	-9	no	0

The answer should be -2 but the code returns 0. Bug found: initialise `best = a[0]` or `float('-inf')` (or just use `max(a)`). A test with only positive numbers would never have caught it. That is why you choose test inputs on purpose.

⚠️ COMMON MISTAKE

Dry running only the happy path. Most real bugs live at the edges: index 0, index n-1, empty input, negatives, duplicates, and very large values. Always test at least one edge case.

🧰 AT WORK

Dry runs are what senior engineers do in code review. When a reviewer writes "what happens if the list is empty here?", they are dry running your code in their head. A classic production bug: a dashboard showing "max latency 0 ms" because the max was initialised to 0 and the day had no traffic. The same habit that wins interviews prevents those incidents.

✓ CHECK YOUR PROGRESS 1.4**1. Why should you dry run the code you wrote, not your idea?**

Answer: Bugs come from the gap between idea and code: wrong boundary, wrong initial value, wrong order of statements. Tracing the idea skips exactly those mistakes.

2. Give three edge cases to test for an array problem.

Answer: Empty array, single element, all elements equal (also: negatives, duplicates, very large values).

1.5 How to Talk While Coding

An interview is a conversation, not an exam hall. The interviewer wants to see how you think, and they can only see what you say. Silence for five minutes looks the same as being stuck.

1.5.1 What to say at each stage

Stage	Example lines
Clarifying	"Can the array be empty? Should I return -1 then?"
Brute force	"The simple way is to try all pairs. That's $O(n^2)$. Let me think if we can do better."
Optimising	"The repeated work is searching for the complement. A dict would make that $O(1)$."
Coding	"I'm storing value to index. I check before inserting so I don't use the same element twice."
Stuck	"I'm unsure how to handle duplicates here. Let me try a small example."
Testing	"Let me trace [3,3] with target 6..."
Wrapping up	"Time $O(n)$, space $O(n)$. If memory were tight, I could sort and use two pointers, but I'd lose original indices."

1.5.2 Handling hints

A hint is not a failure. It is the interviewer helping you. Say "thank you, that helps", repeat the hint in your words, and use it. Arguing with a hint, or ignoring it, is a strong negative signal.

INTERVIEW CORNER

Q: What do you do if you are completely stuck?

A: Say so calmly, go back to a small example and solve it by hand, and mention the patterns you are considering ("Could this be a sliding window? The window condition isn't monotonic, so maybe not."). Thinking aloud often triggers a useful hint.

Q: Should I write the optimal solution directly if I know it?

A: Briefly mention the brute force and its complexity first (one sentence), then explain the optimal idea and get agreement before coding. Jumping straight to code can look like memorisation.

Q: How long should I spend before coding?

A: In a 45-minute round, roughly 5 to 10 minutes understanding and discussing, about 15 to 20 minutes coding, and the rest testing and on follow-ups.

✓ CHECK YOUR PROGRESS 1.5**1. Why is silence harmful in a coding round?**

Answer: The interviewer grades your thinking. If you are silent, they cannot tell whether you are making progress or stuck, and they cannot give a timely hint.

2. How should you respond to a hint?

Answer: Acknowledge it, restate it in your own words, and apply it to your approach. Do not argue with it or ignore it.

1.6 Python Essentials for Interviews

Chapter 0 covers Python for DSA in full: numbers, strings, lists, dicts, `collections`, `heapq`, `bisect`, sorting and recursion. This section is only a one-page reminder of the tools you will reach for in every chapter. If any line below feels unfamiliar, go back to Chapter 0 before moving on.

1.6.1 The everyday toolkit

Need	Python tool	Key operations	Time
Dynamic array	list	append, pop(), a[i], len	O(1) (append amortised); pop(0) and insert(0, x) are O(n)
Key → value	dict / defaultdict / Counter	d[k], d.get(k, 0), k in d, Counter(s)	O(1) average
Unique items	set	add, in, remove, discard	O(1) average
Stack	list	append, pop(), a[-1]	O(1)
Queue / deque	collections.deque	append, popleft, appendleft, pop	O(1)
Min-heap	heapq on a list	heappush, heappop, h[0]	push/pop O(log n), peek O(1)

Need	Python tool	Key operations	Time
Max-heap	heapq with negated values	heappush(h, -x), -heappop(h)	$O(\log n)$
Search in sorted list	bisect	bisect_left, bisect_right	$O(\log n)$

```

from collections import Counter, defaultdict, deque
import heapq

freq = Counter("banana")                # Counter({'a': 3, 'n': 2, 'b': 1})
groups = defaultdict(list)
groups["key"].append("word")            # no KeyError on first use

q = deque([(0, 0)])                     # BFS queue of grid cells
r, c = q.popleft()                     # O(1); never use list.pop(0)

h = []
heapq.heappush(h, (5, "task"))          # tuples compare by first item
pairs = sorted([[5, 6], [1, 3]], key=lambda p: p[0]) # sort by start
s = "".join(["a", "b", "c"])           # build strings with join, not += in a loop

```

⚠ COMMON MISTAKE

Three Python traps cost the most marks: using `list.pop(0)` as a queue ($O(n)$, use `deque`), building a grid with `[[0] * m] * n` (every row is the same list, use `[[0] * m for _ in range(n)]`), and deep recursion on 10^5 nodes (the default limit is about 1000; raise it with `sys.setrecursionlimit` or write the loop iteratively). Chapter 0 covers these in detail.

🔑 KEY POINT

Python ints never overflow, so you do not need `long` casts. But if the problem says "return the answer modulo $10^9 + 7$ ", still take `% MOD` after each step: it keeps numbers small and fast, and the judge expects it.

✅ CHECK YOUR PROGRESS 1.6

1. Which Python structure should you use for a BFS queue, and why not a plain list?

Answer: `collections.deque` with `append` and `popleft`, both $O(1)$. `list.pop(0)` shifts every element, so it costs $O(n)$.

2. How do you get a max-heap in Python?

Answer: `heapq` only gives a min-heap, so push `-x` and negate again when you pop.

What Have We Learnt?

- Use the seven steps: understand, examples, brute force, optimise, code, test, analyse.
- Clarifying questions about empty input, negatives, duplicates, sortedness and size change the solution.
- State the brute force with its complexity before optimising.
- Optimise by removing bottlenecks, unnecessary work and duplicated work (hashing, sorting, precomputation, better data structures).
- A judge does roughly 10^8 simple operations per second; use the constraint table to guess the target complexity.
- $n \leq 20$ hints at exponential; n up to 10^6 hints at $O(n \log n)$ or $O(n)$; $n \geq 10^9$ hints at $O(\log n)$.
- About 15 to 20 patterns cover most interview problems; signal words point to them.
- Dry run your actual code with a variable table, including edge cases.
- Talk continuously: clarify, explain, admit when stuck, and use hints gracefully.
- Know list, dict, set, Counter, defaultdict, deque, heapq and bisect by heart (Chapter 0).
- Build strings with `".join`, use deque for queues, and build 2D grids with a comprehension.
- Python ints never overflow, but still apply `% MOD` when asked, and watch float precision and the recursion limit.

Key terms: problem-solving framework, clarifying question, brute force, BUD optimisation, constraint analysis, pattern recognition, dry run, edge case, deque, heapq, dict, Counter, defaultdict, bisect, recursion limit, modulo $10^9 + 7$.

Exercises

A. Multiple Choice Questions

- For $n \leq 10^5$, which complexity is the safest target?
 - $O(n^3)$
 - $O(n^2)$
 - $O(n \log n)$
 - $O(2^n)$
- Which Python structure is best for a FIFO queue?
 - list with `pop(0)`
 - `collections.deque`
 - set
 - tuple
- How do you make `heapq` behave as a max-heap of numbers?
 - `heapq.heapify(h, reverse=True)`
 - Push negated values
 - Sort the list after every push
 - Use `heapq.maxheap()`
- A problem asks for "all permutations" of a list with $n \leq 8$. The pattern is:
 - Sliding window
 - Backtracking
 - Binary search
 - Prefix sum
- After `g = [[0] * 3] * 3; g[0][0] = 1`, what is `g[1][0]`?
 - 0
 - 1
 - `IndexError`
 - None

B. Fill in the Blanks

- The safe midpoint formula used in most languages is `mid = lo + _____`.
- To pop the smallest item from a heap list `h`, call _____.
- Building a string with `+=` inside a loop can cost _____ time in total; use `".join` instead.

C. True or False

- A dry run should trace the idea, not the exact code.
- dict lookup is $O(1)$ on average.
- Python's default recursion limit is high enough for a DFS 10^5 levels deep.

D. Short Answer Questions

- List the seven steps of the framework and one line on each.
- Practice problem: "Contains Duplicate" (return true if any value appears twice). Hint: what data structure answers "have I seen this before?" in $O(1)$?
- Practice problem: "Find the Maximum in an Array" with possibly all-negative values. Hint: think about the initial value.

E. Long Answer Questions

1. Explain how input constraints guide the choice of algorithm, with three worked examples.
2. Apply the full seven-step framework to "First Unique Character in a String" (return the index of the first non-repeating character). Hint: two passes with a Counter or a count list of size 26.

F. Design Challenge

Design your own one-page "interview checklist" card that you will keep next to you while practising. It must cover clarifying questions, the constraint table, five edge cases and three Python pitfalls.

Answer Key: Chapter 1

A. MCQ

1. (c) $n \log n \approx 1.7 \times 10^6$ operations; $n^2 = 10^{10}$ is too slow.
2. (b) deque gives $O(1)$ append and popleft; `list.pop(0)` is $O(n)$.
3. (b) heapq is a min-heap only, so store $-x$ and negate on pop.
4. (b) "All" plus tiny n means backtracking, $O(n \cdot n!)$.
5. (b) All three rows are the same list object, so changing one row changes all of them.

B. Fill in the Blanks

1. $(hi - lo) // 2$ (in Python, $(lo + hi) // 2$ is also safe since ints do not overflow)
2. `heapq.heappop(h)`
3. $O(n^2)$

C. True or False

1. False. Trace the actual code; bugs live in the gap between idea and code.
2. True. Worst case can degrade with many hash collisions, but average is $O(1)$.
3. False. The default limit is about 1000. Raise it with `sys.setrecursionlimit` or use an explicit stack.

D. Short Answers

1. Understand (restate, clarify), Examples (solve by hand, add edge cases), Brute force (simplest correct idea plus complexity), Optimise (remove repeated work), Code (clean, named variables), Test (dry run with edge cases), Analyse (time, space, trade-offs).
2. Approach: iterate with a set; if a value is already in seen, a duplicate exists, else add it. Time $O(n)$, space $O(n)$. One-liner: `len(set(nums)) < len(nums)`. Alternative: sort and compare neighbours, $O(n \log n)$ time, $O(1)$ extra space (if sorting in place is allowed).
3. Approach: start with `best = a[0]` (or `float('-inf')`) and scan, or simply use `max(a)`. Initialising to 0 fails for all-negative arrays. Time $O(n)$, space $O(1)$.

E. Long Answers

1. Model answer:
 - Judges allow roughly 10^8 simple operations per second (Python is slower, so leave extra headroom).
 - Compute the operation count of a candidate complexity using n 's maximum.
 - $n \leq 10^5$: $n^2 = 10^{10}$ too slow, $n \log n \approx 1.7 \times 10^6$ fine, so use sorting, heaps, hashing or two pointers.
 - $n \leq 20$: $2^{20} \approx 10^6$ fine, so subset enumeration or bitmask DP is intended.
 - $n \leq 10^9$: only $O(\log n)$ or $O(1)$ work, so binary search or math.
 - Also check value ranges for float precision and for counting-list tricks.

- Two-dimensional inputs ($m, n \leq 1000$) suggest $O(m \cdot n)$ grid DP or BFS.

2. Model answer:

- Understand: lowercase letters only? Return -1 if none.
- Examples: "leetcode" $\rightarrow$ 0, "loveleetcode" $\rightarrow$ 2, "aabb" $\rightarrow$ -1.
- Brute force: for each i , scan the whole string to count $s[i]$: $O(n^2)$.
- Optimise: first pass builds `count = Counter(s)` ; second pass returns the first i with count 1.
- Code: `for i, ch in enumerate(s): if count[ch] == 1: return i, then return -1.`
- Test: "aabb" returns -1; single "z" returns 0.
- Analyse: $O(n)$ time, $O(1)$ space (at most 26 keys).

F. Design Challenge: Model Solution

A good card has four blocks. (1) Clarify: empty input? negatives? duplicates? sorted? modify input allowed? return value when no answer? (2) Constraints: $10 \rightarrow n!$, $20 \rightarrow 2^n$, $500 \rightarrow n^3$, $5000 \rightarrow n^2$, $10^6 \rightarrow n \log n$, $10^9 \rightarrow \log n$. (3) Edge cases: empty, one element, all equal, negatives and zero, maximum values. (4) Python pitfalls: `list.pop(0)` as a queue, the `[[0] * m] * n` grid, mutable default arguments, and the recursion limit. Add one line at the bottom: "Brute force out loud, then optimise, then ask before coding."