

DSA

Data Structures & Algorithms in Java

Start from Java 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.

● ● ● Solution.java

```
class Solution {  
    Offer crack(Problem p) {  
        Pattern pt = spotPattern(p); // 20 patterns  
        Trick t = pt.keyTrick();      // the one idea  
        return code(t);              //  $O(n)$ , not  $O(n^2)$   
    }  
}
```

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 Java, 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 Java 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 Java 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 Java.

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	Java Basics for DSA (Start Here)	7
1	How to Approach Any DSA Problem	31
2	Time and Space Complexity	48

PART II: LINEAR PATTERNS

3	Arrays, Strings and Prefix Sums	66
4	Two Pointers	82
5	Sliding Window	99
6	Hashing	117
7	Binary Search	135
8	Sorting and Intervals	152

PART III: RECURSION AND LINEAR STRUCTURES

9	Recursion and Backtracking	172
10	Linked Lists	192
11	Stacks, Queues and Monotonic Stack	210

PART IV: TREES, HEAPS AND GRAPHS

12	Trees and Binary Search Trees	228
13	Heaps and Priority Queues	247
14	Graphs I: BFS, DFS and Topological Sort	264
15	Graphs II: Shortest Paths, Union-Find and MST	283

PART V: DYNAMIC PROGRAMMING AND GREEDY

16	Dynamic Programming I: Foundations	301
17	Dynamic Programming II: Grids, Strings and Subsequences	317
18	Greedy Algorithms	337

PART VI: TRICKS AND PRACTICE

19	Tries, Bit Manipulation and Math Tricks	356
20	Must-Do Problem Bank and Pattern Cheat Sheet	374

APPENDICES

A	Pattern Recognition Cheat Sheet	391
B	Big-O and Java Collections Cheat Sheet	394
C	Java Code Templates	397
D	Glossary	403

PART I

Foundations

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

CHAPTER 0

Java Basics for DSA (Start Here)

"You cannot bat well if you keep looking for the bat. Know your tools before the match starts."

After studying this chapter, you will be able to

- Write a correct `class Solution` method and a full program with `main`.
- Choose between `int` and `long`, and spot integer overflow before it bites.
- Use arrays, Strings, `StringBuilder` and char arithmetic without looking anything up.
- Use `ArrayList`, `HashMap`, `HashSet`, `TreeMap`, `TreeSet`, `ArrayDeque`, `PriorityQueue` and `LinkedList` fluently.
- Write comparators with lambdas and avoid the `Integer == trap`.
- Read input fast in online tests and fix the most common compile errors.

You write Python or JavaScript at work. Then the online test says "Java only", and you are searching "sort array descending java" with the timer running. This chapter is a toolbox, not a theory class.

A dabbawala never thinks about holding the crate; his attention goes to the right tiffin. Make Java syntax your muscle memory, so attention goes to the pattern.

0.1 Program Structure and class Solution

On LeetCode-style platforms you only fill in a method inside `class Solution`. On HackerRank-style tests you often write a full program with `main`. Know both.

```
import java.util.*;           // gives ArrayList, HashMap, Arrays, etc.

class Solution {
    public int sum(int[] nums) {           // return type, name, parameters
        int total = 0;
        for (int x : nums) total += x;
        return total;
    }
}

public class Main {                // file must be Main.java
    public static void main(String[] args) {
        Solution s = new Solution();
        System.out.println(s.sum(new int[]{1, 2, 3})); // 6
    }
}
```

- Every statement ends with `;`. Blocks use `{ }`.
- `static` methods belong to the class; you can call them from `main` without creating an object.
- Fields declared in the class act as "global" variables shared by helper methods.

KEY POINT

Always start with `import java.util.*;`. Most "cannot find symbol" errors in tests are a missing import.

✓ CHECK YOUR PROGRESS 0.1

1. Why can `main` call a static helper directly but not a non-static one?

Answer: `main` is static, so there is no object (`this`). A non-static method needs an object, so create one first: `new Solution().helper()`.

2. What must the file be named if it contains `public class Main`?

Answer: `Main.java`. A public class must match its file name.

0.2 Primitive Types, Ranges and Overflow

Type	Size	Range (about)	Use in DSA
<code>int</code>	32 bit	$\pm 2.1 \times 10^9$ (2,147,483,647)	Default for counts, indexes
<code>long</code>	64 bit	$\pm 9.2 \times 10^{18}$	Sums, products, answers mod 10^9+7
<code>double</code>	64 bit	15 to 16 digits precision	Averages, rarely needed
<code>char</code>	16 bit	0 to 65,535	Characters, also a number
<code>boolean</code>	-	true / false	visited[] arrays, flags

0.2.1 Overflow

When an `int` goes past its maximum, it silently wraps around to a negative number. No error, just a wrong answer.

```
int a = 2_000_000_000, b = 2_000_000_000;
int bad = a + b;           // -294967296 (wrapped!)
long good = (long) a + b;   // 4000000000, cast BEFORE adding
long wrong = a * b;         // still overflows: int*int computed first
int mid = lo + (hi - lo) / 2; // safe midpoint for binary search
```

Rule of thumb: if values reach 10^9 and you add or multiply them, use `long`. Constants: `Integer.MAX_VALUE`, `Integer.MIN_VALUE`, `Long.MAX_VALUE`.

⚠ COMMON MISTAKE

`Math.abs(Integer.MIN_VALUE)` is still negative, because $+2,147,483,648$ does not fit in an `int`. Also, `Integer.MAX_VALUE + 1` used as "infinity" can overflow when you add to it. Use `Integer.MAX_VALUE / 2` as a safe infinity for distances.

👛 AT WORK

A report stored totals in paise as `int`. Once a merchant crossed about ₹2.1 crore (2^{31} paise), revenue showed negative. The fix: `long`. Store money as `long` or `BigDecimal`, never `double`.

✅ CHECK YOUR PROGRESS 0.2

1. Is $10^9 + 10^9$ safe in an `int`?

Answer: Just barely: 2×10^9 is under $2,147,483,647$, so it fits. One more addition would overflow, so use `long` to be safe.

2. Why does `long x = a * b;` overflow when `a` and `b` are `int`?

Answer: The multiplication happens in `int` first, then the wrong result is widened. Write `(long) a * b`.

0.3 Operators: Division, Modulo and Friends

Expression	Result	Why
<code>7 / 2</code>	3	<code>int</code> division drops the decimal
<code>-7 / 2</code>	-3	rounds towards zero, not down
<code>7 / 2.0</code>	3.5	one double makes it double division
<code>-7 % 3</code>	-1	sign follows the left operand
<code>Math.floorMod(-7, 3)</code>	2	always non-negative for positive divisor

Expression	Result	Why
$(a + b - 1) / b$	$\text{ceil}(a/b)$	ceiling for positive ints
$x \& 1$	0 or 1	odd check (works for negatives too)
$1 \ll k$	2^k	left shift

`i++` uses the value then increments; `++i` increments first. `+=` hides a cast: `int x = 0; x += 2.7;` gives 2.

⚠ COMMON MISTAKE

Checking odd with `x % 2 == 1` fails for negative numbers (`-3 % 2` is `-1`). Use `x % 2 != 0`. For circular arrays with negative steps, use `((i % n) + n) % n`.

✅ CHECK YOUR PROGRESS 0.3

1. What is `-10 % 4` in Java?

Answer: -2. The sign follows the dividend. `Math.floorMod(-10, 4)` gives 2.

2. How do you compute the number of pages needed for 23 items with 5 per page?

Answer: $(23 + 5 - 1) / 5 = 27 / 5 = 5$.

0.4 Control Flow and Methods

```
if (x > 0) { ... } else if (x == 0) { ... } else { ... }
String kind = (x % 2 == 0) ? "even" : "odd";           // ternary

for (int i = 0; i < n; i++) { ... }                  // index loop
for (int i = n - 1; i ≥ 0; i--) { ... }              // backwards
for (int x : nums) { ... }                          // for-each, read only
while (lo ≤ hi) { ... }                             // unknown count

outer:
for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++)
        if (grid[i][j] == 0) break outer;           // break both loops
```

Methods: Java passes everything **by value**. For objects (arrays, lists), the value passed is the reference, so changes to the contents are visible to the caller, but reassigning the parameter is not.

```
void change(int[] arr, int x) {
    arr[0] = 99;    // caller sees this
    x = 5;          // caller does NOT see this
    arr = new int[3]; // caller does NOT see this either
}
```

KEY POINT

Need to "return" two values from a helper? Return an `int[]` like `new int[]{min, max}`, or update a class field.

CHECK YOUR PROGRESS 0.4

1. Can you modify elements inside a for-each loop over an `int` array?

Answer: No. `x` is a copy; assigning to it does not change the array. Use an index loop.

2. How do you exit two nested loops at once?

Answer: Use a labelled `break` (`break outer;`), or put the loops in a method and `return`.

0.5 Arrays

```
int[] a = new int[5];           // all 0
int[] b = {3, 1, 2};
int n = b.length;              // field, no brackets
int[][] grid = new int[3][4];  // 3 rows, 4 columns
int rows = grid.length, cols = grid[0].length;
boolean[] seen = new boolean[n]; // all false
String[] names = new String[2]; // all null

Arrays.fill(a, -1);            // set all to -1
for (int[] row : grid) Arrays.fill(row, Integer.MAX_VALUE); // 2D fill
Arrays.sort(b);                // [1, 2, 3]
Arrays.sort(b, 0, 2);          // sort index 0..1 only
System.out.println(Arrays.toString(b)); // [1, 2, 3]
System.out.println(Arrays.deepToString(grid)); // 2D print
int[] copy = Arrays.copyOf(b, b.length);
int[] part = Arrays.copyOfRange(b, 1, 3); // index 1..2
int idx = Arrays.binarySearch(b, 2); // array must be sorted
```

An `int[]` cannot take a comparator. For descending order, sort ascending and read backwards, or use `Integer[]`.

DO YOU KNOW?

Since Java 7, `Arrays.sort` on primitive arrays uses Dual-Pivot Quicksort, while sorting objects uses TimSort, an algorithm first written by Tim Peters for Python.

COMMON MISTAKE

Printing an array directly gives something like `[I@1b6d3586`. Use `Arrays.toString`. And `int[] c = b;` does not copy; both names point to the same array.

✓ CHECK YOUR PROGRESS 0.5

1. Why does `Arrays.fill(grid, 0)` fail on a 2D int array?

Answer: A 2D array is an array of row arrays. You must fill each row: `for (int[] row : grid) Arrays.fill(row, 0);`

2. What is the default value of a new `int[]`, `boolean[]` and `String[]`?

Answer: 0, false and null.

0.6 Strings, StringBuilder and char Arithmetic

0.6.1 String API

Method	Example	Cost
<code>length()</code>	<code>"abc".length() → 3</code>	$O(1)$
<code>charAt(i)</code>	<code>"abc".charAt(1) → 'b'</code>	$O(1)$
<code>substring(i, j)</code>	<code>"hello".substring(1, 3) → "el"</code> (j excluded)	$O(j - i)$
<code>toCharArray()</code>	<code>char[] cs = s.toCharArray();</code>	$O(n)$
<code>equals(t)</code>	<code>s.equals(t)</code> , never <code>s == t</code>	$O(n)$
<code>compareTo(t)</code>	<code>"apple".compareTo("bat") < 0</code>	$O(n)$
<code>indexOf(x)</code>	<code>"banana".indexOf("an") → 1</code>	$O(n \cdot m)$
<code>split(regex)</code>	<code>"a b".split(" ")</code>	$O(n)$
<code>String.valueOf(x)</code>	int or <code>char[]</code> to String	$O(n)$
<code>Integer.parseInt(s)</code>	<code>"42" → 42</code>	$O(n)$

0.6.2 String vs StringBuilder

A String is **immutable**: every `s += c` builds a brand new String and copies everything. In a loop of n steps that is $O(n^2)$. StringBuilder is a growable buffer, so appends are amortised $O(1)$.

```
StringBuilder sb = new StringBuilder();
for (int i = 0; i < n; i++) sb.append(i).append(',');
sb.setLength(sb.length() - 1); // drop last comma
sb.reverse();                  // in place
sb.deleteCharAt(sb.length() - 1); // backtracking "undo"
sb.setCharAt(0, 'y');
String result = sb.toString();
```

0.6.3 char Arithmetic

A char is a small number (its Unicode value). That makes counting letters easy.

```
int idx = c - 'a';           // 'a'→0, 'z'→25
char next = (char) (c + 1);  // 'a' → 'b' (cast needed)
int digit = c - '0';         // '7' → 7
char up = Character.toUpperCase(c);
Character.isDigit(c); Character.isLetter(c); Character.isLetterOrDigit(c);
int[] freq = new int[26];
for (char ch : s.toCharArray()) freq[ch - 'a']++;
```

😄 FUNNY SIDE

Writing `s += ch` inside a loop of 10^5 is like rewriting your whole shopping list on a new sheet every time you remember one more item. Your mother will notice. So will the time limit.

⚠️ COMMON MISTAKE

`s1 == s2` compares references, not text. It may "work" for literals (string pool) and then fail for strings built at runtime. Always use `equals`. Also, `'a' + 1` is the int 98, and `"" + 'a' + 1` is the String "a1".

✅ CHECK YOUR PROGRESS 0.6

1. What does `"interview".substring(2, 5)` return?

Answer: "ter" (indexes 2, 3, 4; the end index is excluded).

2. Why use `StringBuilder` when building a string in a loop?

Answer: Strings are immutable, so each concatenation copies the whole string, giving $O(n^2)$. `StringBuilder` appends in amortised $O(1)$, so the loop is $O(n)$.

0.7 Wrapper Classes and Autoboxing

Collections hold **objects**, not primitives. So Java has a wrapper for each: `Integer` for `int`, `Long`, `Character`, `Boolean`, `Double`. Converting automatically between them is called autoboxing (`int` → `Integer`) and unboxing (`Integer` → `int`).

```
List<Integer> list = new ArrayList<>(); // not List<int>
list.add(5);                       // autobox 5 → Integer
int x = list.get(0);                // unbox

Integer a = 127, b = 127;
System.out.println(a == b);         // true (cached)
Integer c = 1000, d = 1000;
System.out.println(c == d);         // false (different objects!)
System.out.println(c.equals(d));    // true
```

Java caches `Integer` objects from -128 to 127, so `==` works for small numbers. Your samples pass, then hidden tests with large values fail.

⚠ COMMON MISTAKE

`if (map.get(a) == map.get(b))` compares two `Integer` objects. Use `map.get(a).equals(map.get(b))`, or unbox into `int` variables first. Also, unboxing `null` throws `NullPointerException`: `int v = map.get(missingKey);` crashes.

🔗 INTERVIEW CORNER

Q: Why does `Integer a = 128, b = 128; a == b` give false?

A: `==` on objects compares references. Autoboxing uses `Integer.valueOf`, which returns cached objects only for `-128` to `127`. `128` creates two separate objects.

✅ CHECK YOUR PROGRESS 0.7

1. Why can you not write `List<int>`?

Answer: Generics work only with object types. Use the wrapper `List<Integer>`.

2. What happens on `int v = map.get("x");` if `"x"` is missing?

Answer: `get` returns `null`, and unboxing `null` throws a `NullPointerException`. Use `getOrDefault("x", 0)`.

0.8 The Collections Toolbox

Declare with the interface on the left and the class on the right: `List<Integer> list = new ArrayList<>();`. The `<>` (diamond) lets Java infer the type.

Need	Java class	Key ops cost
Growable array	<code>ArrayList</code>	<code>get/set</code> $O(1)$, <code>add</code> at end $O(1)^*$
Key $\rightarrow$ value lookup	<code>HashMap</code>	<code>get/put</code> $O(1)$ average
Unique items, membership	<code>HashSet</code>	<code>add/contains</code> $O(1)$ average
Sorted keys, floor/ceiling	<code>TreeMap</code> / <code>TreeSet</code>	$O(\log n)$
Stack or queue	<code>ArrayDeque</code>	<code>push/pop/offer/poll</code> $O(1)$
Always get min/max	<code>PriorityQueue</code>	<code>offer/poll</code> $O(\log n)$, <code>peek</code> $O(1)$
Insert/remove at ends, list ops	<code>LinkedList</code>	<code>ends</code> $O(1)$, <code>get(i)</code> $O(n)$

** amortised: occasionally the array doubles, but on average each add is $O(1)$.*

0.8.1 ArrayList

Method	Meaning	Cost
<code>add(x)</code> / <code>add(i, x)</code>	append / insert at <code>i</code>	$O(1)^*$ / $O(n)$

Method	Meaning	Cost
get(i) / set(i, x)	read / write	O(1)
remove(int i) / remove(Object o)	by index / by value	O(n)
size(), isEmpty(), contains(x)	info	O(1), O(1), O(n)

```
List<Integer> list = new ArrayList<>(List.of(5, 3, 8));
list.remove(Integer.valueOf(3)); // removes VALUE 3
list.remove(0);                 // removes INDEX 0
List<List<Integer>> result = new ArrayList<>();
result.add(new ArrayList<>(list)); // copy! not result.add(list)
```

⚠ COMMON MISTAKE

In backtracking, adding `list` itself to the result stores a reference that keeps changing. Always add a copy: `new ArrayList<>(list)`. Also, `List.of(...)` is immutable; calling add on it throws an exception.

0.8.2 HashMap

Method	Meaning
put(k, v) / get(k)	insert or overwrite / value or null
getOrDefault(k, d)	value, or d if missing
containsKey(k)	is key present
merge(k, 1, Integer::sum)	add 1 to count (insert 1 if missing)
computeIfAbsent(k, x -> new ArrayList<>())	get list, creating if missing
remove(k), size()	delete key, count keys
keySet(), values(), entrySet()	views for looping

```
Map<String, Integer> count = new HashMap<>();
count.put("chai", count.getOrDefault("chai", 0) + 1);
count.merge("chai", 1, Integer::sum); // same thing, shorter
Map<Integer, List<String>> groups = new HashMap<>();
groups.computeIfAbsent(4, k -> new ArrayList<>()).add("tea");
for (Map.Entry<String, Integer> e : count.entrySet())
    System.out.println(e.getKey() + " = " + e.getValue());
```

HashMap has no order. Use `LinkedHashMap` to keep insertion order and `TreeMap` for sorted keys.

0.8.3 HashSet

```
Set<Integer> seen = new HashSet<>();
boolean isNew = seen.add(4); // true first time, false if already there
seen.contains(4);           // true
seen.remove(4);
Set<String> visited = new HashSet<>();
visited.add(r + "," + c);    // grid cell as a key (or use r * cols + c)
```

0.8.4 TreeMap and TreeSet

These keep keys sorted (a red-black tree inside). Use them when you need "the nearest key".

Method	Returns
<code>floor(x) / floorKey(x)</code>	largest $\leq x$, or null
<code>ceiling(x) / ceilingKey(x)</code>	smallest $\geq x$, or null
<code>lower(x) / higher(x)</code>	strictly $< x$ / strictly $> x$
<code>first() / last()</code> (firstKey/lastKey)	min / max
<code>pollFirst() / pollFirstEntry()</code>	remove and return min
<code>headSet(x) / tailMap(x)</code>	views below / from x

```
TreeSet<Integer> ts = new TreeSet<>(List.of(10, 20, 30));
ts.floor(25);      // 20
ts.ceiling(25);    // 30
ts.floor(5);       // null, check before unboxing!
TreeMap<Integer, String> slots = new TreeMap<>();
slots.put(9, "standup"); slots.put(14, "review");
slots.floorEntry(11).getValue(); // "standup"
```

0.8.5 ArrayDeque as Stack and Queue

Use	Add	Remove	Look
Stack (LIFO)	<code>push(x)</code>	<code>pop()</code>	<code>peek()</code>
Queue (FIFO)	<code>offer(x)</code>	<code>poll()</code>	<code>peek()</code>
Both ends	<code>offerFirst / offerLast</code>	<code>pollFirst / pollLast</code>	<code>peekFirst / peekLast</code>

```
Deque<Integer> stack = new ArrayDeque<>();
stack.push(1); stack.push(2);
stack.pop();         // 2
Queue<int[]> q = new ArrayDeque<>();    // BFS queue of cells
q.offer(new int[]{0, 0});
int[] cell = q.poll();    // {0, 0}
```

Avoid the old `Stack` class (it is synchronised and slower). `ArrayDeque` does not allow null elements. `pop()` on an empty deque throws; `poll()` returns null.

0.8.6 PriorityQueue (Heap)

```
PriorityQueue<Integer> minHeap = new PriorityQueue<>();
PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder());
PriorityQueue<int[]> byDist = new PriorityQueue<>((a, b) → a[1] - b[1]); // [node, dist]
minHeap.offer(5); minHeap.offer(1); minHeap.offer(3);
minHeap.peek();    // 1
minHeap.poll();    // 1, now min is 3
```

Printing or iterating a `PriorityQueue` is **not** sorted; only repeated `poll()` is. `remove(x)` is $O(n)$.

0.8.7 LinkedList

`LinkedList` implements `List` and `Deque`: `addFirst`, `addLast`, `removeFirst`, `removeLast` are $O(1)$, but `get(i)` is $O(n)$. `ArrayDeque` is usually faster; use `LinkedList` when you need null elements.

INTERVIEW CORNER

Q: `HashMap` vs `TreeMap`?

A: `HashMap`: $O(1)$ average, no order. `TreeMap`: $O(\log n)$, keys sorted, gives floor/ceiling and range queries. Choose `TreeMap` only when you need order.

Q: Is `HashMap` really $O(1)$?

A: On average. Since Java 8, crowded buckets become balanced trees, so the worst case is about $O(\log n)$.

CHECK YOUR PROGRESS 0.8

1. On `List<Integer> list = [5, 3, 8]`, what does `list.remove(1)` do?

Answer: Removes the element at index 1 (value 3). To remove the value 1, write `list.remove(Integer.valueOf(1))`.

2. Which structure gives "the largest booked time $\leq t$ " quickly?

Answer: A `TreeSet` or `TreeMap`, using `floor(t)` or `floorKey(t)` in $O(\log n)$.

0.9 Comparators, Lambdas and Collections Methods

A comparator returns negative if `a` comes first, positive if `b` comes first, 0 if equal. A lambda `(a, b) → ...` writes one in a line.

```
Integer[] arr = {3, 1, 2};
Arrays.sort(arr, (a, b) → b - a);           // descending (small values only)
Arrays.sort(arr, Collections.reverseOrder()); // descending, safe

int[][] intervals = {{5, 6}, {1, 3}, {2, 4}};
Arrays.sort(intervals, (a, b) → Integer.compare(a[0], b[0])); // by start

List<String> words = new ArrayList<>(List.of("kiwi", "fig", "apple"));
words.sort(Comparator.comparingInt(String::length)
    .thenComparing(Comparator.naturalOrder())); // length, then alphabet
```

⚠ COMMON MISTAKE

$(a, b) \rightarrow a - b$ overflows when values are large or of mixed sign (e.g. $a = 2 \times 10^9$, $b = -2 \times 10^9$).
Use `Integer.compare(a, b)` or `Long.compare` whenever values can be big.

Collections method	Does
<code>Collections.sort(list)</code>	sort ascending (stable)
<code>Collections.reverse(list)</code>	reverse in place
<code>Collections.max(c) / min(c)</code>	largest / smallest
<code>Collections.swap(list, i, j)</code>	swap two items
<code>Collections.frequency(c, x)</code>	count of x, $O(n)$
<code>Collections.reverseOrder()</code>	descending comparator

✓ CHECK YOUR PROGRESS 0.9

1. Write a comparator that sorts `int[]` pairs by second value descending.

Answer: $(a, b) \rightarrow \text{Integer.compare}(b[1], a[1])$.

2. Why does `Arrays.sort(intArr, Collections.reverseOrder())` not compile?

Answer: Comparators work only on object arrays. `int[]` is primitive; use `Integer[]` or sort ascending and reverse.

0.10 Recursion Basics

A recursive method calls itself on a smaller input. Every recursion needs two parts: a **base case** that stops, and a **recursive step** that moves towards it. Think of Russian dolls: open one, find a smaller one, stop at the solid doll.

```
int factorial(int n) {
    if (n ≤ 1) return 1;           // base case
    return n * factorial(n - 1);   // smaller problem
}
```

```
factorial(3)
= 3 * factorial(2)
    = 2 * factorial(1)
        = 1          (base case)
    = 2
= 6
```

Figure 0.1: Each call waits on the call stack until the smaller one returns.

Each call uses stack memory, so depth n means $O(n)$ space. Recursion around 10^5 deep (say, DFS on a long chain) can throw `StackOverflowError`; switch to an explicit stack.

🔑 KEY POINT

Before writing a recursive function, say out loud: "What does this function return for input n ?" and "What is the smallest input?" If you can answer both, the code writes itself.

✅ CHECK YOUR PROGRESS 0.10

1. What happens if a recursive method has no base case?

Answer: It calls itself forever until the call stack fills and Java throws `StackOverflowError`.

2. What is the space complexity of recursive `factorial(n)`?

Answer: $O(n)$, because n frames sit on the call stack at the deepest point.

0.11 Fast Input and Common Compile Errors

0.11.1 Reading input fast

`Scanner` is easy but slow for large input (10^5 numbers or more). Use `BufferedReader` with `StringTokenizer`, and `PrintWriter` for output.

```
import java.io.*;
import java.util.*;

public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        PrintWriter out = new PrintWriter(new BufferedWriter(new
OutputStreamWriter(System.out)));
        int n = Integer.parseInt(br.readLine().trim());
        StringTokenizer st = new StringTokenizer(br.readLine());
        long sum = 0;
        for (int i = 0; i < n; i++) sum += Long.parseLong(st.nextToken());
        out.println(sum);
        out.flush();          // forget this and nothing prints!
    }
}
```

 ACTIVITY 0.1

Feed 10^6 numbers to this program and to a Scanner version (`java Main < input.txt`). Time both.

0.11.2 Common compile errors

Error message	Usual cause	Fix
cannot find symbol	Missing import, typo, variable out of scope	Add <code>import java.util.*;</code> , check spelling and braces
incompatible types: possible lossy conversion from long to int	Assigning long to int	Cast <code>(int)</code> or change the variable to long
missing return statement	Some path has no return	Add a final return after loops
int cannot be dereferenced	Calling a method on a primitive, e.g. <code>x.equals(y)</code>	Use <code>==</code> for primitives
non-static method cannot be referenced from a static context	Calling instance method from main	Create an object or make the method static
unreported exception IOException	Using <code>readLine</code> without handling	Add <code>throws IOException</code> to main

 CHECK YOUR PROGRESS 0.11

1. Your program ran but printed nothing. You used `PrintWriter`. Why?

Answer: The output is buffered and was never flushed. Call `out.flush()` (or `out.close()`) at the end.

2. What does "possible lossy conversion from double to int" mean for `int r = Math.sqrt(n);`?

Answer: `Math.sqrt` returns double, which may not fit an int exactly. Write `int r = (int) Math.sqrt(n);`.

0.12 Fifteen Warm-up Problems

Each uses one tool from this chapter. Type them out. All methods sit inside `class Solution`.

0.12.1 Reverse an Array (Easy)

[1, 2, 3, 4] → [4, 3, 2, 1], in place. **Key insight:** two pointers swap from both ends and meet in the middle.

Step	<code>l, r</code>	Array
1	0, 3	[4, 2, 3, 1]
2	1, 2	[4, 3, 2, 1]

Step	l, r	Array
3	2, 1	stop ($l \geq r$)

```
void reverse(int[] a) {
    int l = 0, r = a.length - 1;
    while (l < r) {
        int t = a[l]; a[l] = a[r]; a[r] = t; // swap
        l++; r--;
    }
}
```

Complexity: $O(n)$ time, $O(1)$ space.

0.12.2 Maximum in an Array (Easy)

```
int max(int[] a) {
    int best = Integer.MIN_VALUE; // works even if all are negative
    for (int x : a) best = Math.max(best, x);
    return best;
}
```

Complexity: $O(n)$, $O(1)$. Starting from 0 instead of `MIN_VALUE` fails on `[-5, -2]`.

0.12.3 Count Vowels (Easy)

```
int countVowels(String s) {
    int count = 0;
    for (int i = 0; i < s.length(); i++)
        if ("aeiouAEIOU".indexOf(s.charAt(i)) ≥ 0) count++; // membership trick
    return count;
}
```

Complexity: $O(n)$. "Interview" $\rightarrow 4$.

0.12.4 Palindrome Check (Easy)

Ignore case and non-alphanumeric characters. "A man, a plan, a canal: Panama" $\rightarrow$ true.

```
boolean isPalindrome(String s) {
    int l = 0, r = s.length() - 1;
    while (l < r) {
        char a = s.charAt(l), b = s.charAt(r);
        if (!Character.isLetterOrDigit(a)) { l++; continue; }
        if (!Character.isLetterOrDigit(b)) { r--; continue; }
        if (Character.toLowerCase(a) ≠ Character.toLowerCase(b)) return false;
        l++; r--;
    }
    return true;
}
```

Complexity: $O(n)$, $O(1)$. Empty string $\rightarrow$ true.

0.12.5 Letter Frequency (Easy)

Are two lowercase strings anagrams? "listen", "silent" → true. **Key insight:** an `int[26]` indexed by `c - 'a'`.

```
boolean isAnagram(String s, String t) {
    if (s.length() != t.length()) return false;
    int[] freq = new int[26];
    for (int i = 0; i < s.length(); i++) {
        freq[s.charAt(i) - 'a']++;
        freq[t.charAt(i) - 'a']--;
    }
    for (int f : freq) if (f != 0) return false;
    return true;
}
```

Complexity: $O(n)$, $O(1)$ (26 is constant).

0.12.6 Word Frequency Map (Easy)

```
Map<String, Integer> wordCount(String text) {
    Map<String, Integer> freq = new HashMap<>();
    for (String w : text.trim().toLowerCase().split("\\s+"))
        if (!w.isEmpty()) freq.merge(w, 1, Integer::sum); // count +1
    return freq;
}
```

Complexity: $O(\text{total characters})$. "chai chai coffee" → {chai=2, coffee=1}.

0.12.7 Contains Duplicate (Easy)

```
boolean containsDuplicate(int[] nums) {
    Set<Integer> seen = new HashSet<>();
    for (int x : nums)
        if (!seen.add(x)) return true; // add returns false if present
    return false;
}
```

Brute force: compare every pair, $O(n^2)$. **Complexity:** $O(n)$ time, $O(n)$ space.

0.12.8 Two Sum (Easy)

Return indexes of two numbers adding to target. [2, 7, 11, 15], 9 → [0, 1]. **Key insight:** for each `x`, look up `target - x` in a map of value → index.

```
int[] twoSum(int[] nums, int target) {
    Map<Integer, Integer> idx = new HashMap<>();
    for (int i = 0; i < nums.length; i++) {
        Integer j = idx.get(target - nums[i]); // Integer, may be null
        if (j != null) return new int[]{j, i};
        idx.put(nums[i], i);
    }
    return new int[0];
}
```

Complexity: $O(n)$, $O(n)$. Putting after checking handles [3, 3], 6 correctly.

0.12.9 Reverse Words (Medium)

"the sky is blue" → "blue is sky the".

```
String reverseWords(String s) {
    String[] parts = s.trim().split("\\s+"); // splits on any run of spaces
    StringBuilder sb = new StringBuilder();
    for (int i = parts.length - 1; i ≥ 0; i--) {
        sb.append(parts[i]);
        if (i > 0) sb.append(' ');
    }
    return sb.toString();
}
```

Complexity: $O(n)$. `trim()` avoids an empty first word.

0.12.10 Valid Parentheses (Easy)

"([{}])" → true, "(" → false. **Key insight:** the latest open bracket must close first, so use a stack.

```
boolean isValid(String s) {
    Deque<Character> st = new ArrayDeque<>();
    for (char c : s.toCharArray()) {
        if (c == '(' || c == '[' || c == '{') { st.push(c); continue; }
        if (st.isEmpty()) return false;
        char open = st.pop();
        if ((c == ')' && open != '(') || (c == ']' && open != '[')
            || (c == '}' && open != '{')) return false;
    }
    return st.isEmpty(); // leftover opens mean invalid
}
```

Complexity: $O(n)$, $O(n)$.

0.12.11 Kth Largest Element (Medium)

[3, 2, 1, 5, 6, 4], $k = 2$ → 5. **Key insight:** keep a min-heap of size k ; its top is the k th largest.

```
int findKthLargest(int[] nums, int k) {
    PriorityQueue<Integer> heap = new PriorityQueue<>();
    for (int x : nums) {
        heap.offer(x);
        if (heap.size() > k) heap.poll(); // drop the smallest
    }
    return heap.peek();
}
```

Brute force: sort, $O(n \log n)$. **Complexity:** $O(n \log k)$, $O(k)$.

0.12.12 Sort Meetings by Start (Easy)

Can one person attend all meetings? $[[0, 30], [5, 10]] \rightarrow \text{false}$.

```
boolean canAttend(int[][] meetings) {
    Arrays.sort(meetings, (a, b) → Integer.compare(a[0], b[0]));
    for (int i = 1; i < meetings.length; i++)
        if (meetings[i][0] < meetings[i - 1][1]) return false; // overlap
    return true;
}
```

Complexity: $O(n \log n)$, sorting dominates.

0.12.13 Closest Smaller-or-Equal Price (Easy)

Given prices and a budget per query, return the costliest affordable price, or -1.

```
int[] bestUnder(int[] prices, int[] budgets) {
    TreeSet<Integer> set = new TreeSet<>();
    for (int p : prices) set.add(p);
    int[] ans = new int[budgets.length];
    for (int i = 0; i < budgets.length; i++) {
        Integer f = set.floor(budgets[i]); // may be null
        ans[i] = (f == null) ? -1 : f;
    }
    return ans;
}
```

Complexity: $O((n + q) \log n)$. prices $[10, 25, 40]$, budgets $[30, 5] \rightarrow [25, -1]$.

0.12.14 Power with Recursion (Medium)

Compute $x^n \bmod 10^9+7$ quickly. **Key insight:** $x^n = (x^{(n/2)})^2$, times x if n is odd. Halving gives $O(\log n)$ depth.

```
long power(long x, long n, long mod) {
    if (n == 0) return 1 % mod; // base case
    long half = power(x, n / 2, mod);
    long res = half * half % mod; // long avoids overflow
    if (n % 2 == 1) res = res * (x % mod) % mod;
    return res;
}
```

Dry run: $\text{power}(2, 5) \rightarrow \text{half} = \text{power}(2, 2) = 4 \rightarrow 4 \times 4 = 16 \rightarrow \text{odd}, 16 \times 2 = 32$. **Complexity:** $O(\log n)$ time and stack.

0.12.15 Transpose a Matrix (Easy)

$[[1, 2, 3], [4, 5, 6]] \rightarrow [[1, 4], [2, 5], [3, 6]]$.

```
int[][] transpose(int[][] m) {
    int rows = m.length, cols = m[0].length;
    int[][] t = new int[cols][rows]; // swapped sizes
    for (int r = 0; r < rows; r++)
        for (int c = 0; c < cols; c++)
            t[c][r] = m[r][c];
    return t;
}
```

Complexity: $O(\text{rows} \times \text{cols})$.

INTERVIEW CORNER

Q: In Two Sum, why not sort and use two pointers?

A: Sorting loses the original indexes and costs $O(n \log n)$. The HashMap gives $O(n)$ in one pass. Mention both, then pick.

Q: Why a min-heap for the kth *largest*?

A: The heap keeps the k largest seen so far; the smallest of them, at the top, is the kth largest. Memory is $O(k)$, which matters for streams.

CHECK YOUR PROGRESS 0.12

1. In Two Sum, why is `j` declared as Integer rather than int?

Answer: `get` returns null for a missing key; unboxing null into int would throw `NullPointerException`.

2. Why must the stack be empty at the end of Valid Parentheses?

Answer: Leftover open brackets were never closed, e.g. `"(("` passes the loop but is invalid.

What Have We Learnt?

- Fill a method inside `class Solution`; start every file with `import java.util.*; .`
- `int` holds about $\pm 2.1 \times 10^9$; use `long` for big sums and products, and cast before the operation.
- Integer division truncates towards zero; `%` keeps the sign of the left side; use `Math.floorMod` for a positive result.
- Java passes by value; array contents can be changed by a method, the reference cannot.
- Arrays: `Arrays.fill`, `Arrays.sort`, `Arrays.toString`, and fill 2D arrays row by row.
- Strings are immutable; build strings with `StringBuilder`; compare with `equals`.
- `c - 'a'` maps letters to 0 to 25 for counting arrays.
- Never compare Integer objects with `==`; the cache only covers -128 to 127.
- `HashMap/HashSet` for $O(1)$ lookups; `TreeMap/TreeSet` for order, floor and ceiling.
- `ArrayDeque` is the stack and the queue; `PriorityQueue` is the heap (min by default).
- Use `Integer.compare` in comparators to avoid overflow.
- Recursion needs a base case; deep recursion can cause `StackOverflowError`.
- Use `BufferedReader`, `StringTokenizer` and `PrintWriter` (with `flush`) for large input.

Key terms: `class Solution`, primitive type, overflow, integer division, `floorMod`, pass by value, immutable String, `StringBuilder`, autoboxing, Integer cache, generics, `ArrayList`, `HashMap`, `HashSet`, `TreeMap`, `TreeSet`, `ArrayDeque`, `PriorityQueue`, `LinkedList`, comparator, lambda, base case, `StackOverflowError`, `BufferedReader`.

Exercises

A. Multiple Choice Questions

- What is `-9 / 2` in Java?
 - 5
 - 4
 - 4.5
 - 4
- Which is the best class for a stack in modern Java?
 - Stack
 - Vector
 - ArrayDeque
 - TreeSet
- Integer `a = 200`, `b = 200`; `a == b` is
 - true
 - false
 - compile error
 - NullPointerException
- Which returns the largest element $\leq x$ in a `TreeSet`?
 - `ceiling(x)`
 - `lower(x)`
 - `floor(x)`
 - `higher(x)`
- Default ordering of `new PriorityQueue<Integer>()` is
 - max-heap
 - insertion order
 - min-heap
 - random

B. Fill in the Blanks

- `'d' - 'a'` equals _____.
- To print an int array, use _____.
- In a `HashMap`, `map._____ (k, 1, Integer::sum)` increments a count.

C. True or False

- Iterating over a `PriorityQueue` gives elements in sorted order.
- `long x = (long) a * b;` avoids int overflow.
- `s.substring(1, 4)` includes the character at index 4.

D. Short Answer Questions

- Why is `(a, b) → a - b` a risky comparator?
- When would you choose `TreeMap` over `HashMap`?
- Explain why `result.add(path)` is a bug in backtracking.

E. Long Answer Questions

- Compare `String` and `StringBuilder` with examples and complexity.
- List the Java collections used in DSA and state when to use each, with main operation costs.

F. Design Challenge

Practice set (with hints): (1) **First Unique Character**: count with `int[26]`, then scan again. (2) **Move Zeroes**: write pointer for non-zeros. (3) **Group Anagrams**: key = sorted chars, `computeIfAbsent`. (4) **Implement Queue using Stacks**: two `ArrayDeque`s. (5) **Top K Frequent Elements**: `HashMap` + min-heap of size `k`. Write each in Java inside `class Solution` and state complexity.

Answer Key: Chapter 0

A. MCQ

1. (b) Division truncates towards zero.
2. (c) ArrayDeque is fast and not synchronised.
3. (b) 200 is outside the -128 to 127 cache, so two objects.
4. (c) floor gives largest $\leq x$.
5. (c) Natural ordering, smallest on top.

B. Fill in the Blanks

1. 3
2. Arrays.toString(arr)
3. merge

C. True or False

1. False. Only repeated poll() returns sorted order; the internal array is a heap.
2. True. The cast makes the multiplication happen in long.
3. False. The end index is excluded; it returns indexes 1, 2, 3.

D. Short Answers

1. For large or mixed-sign values, $a - b$ can exceed the int range and wrap, giving the wrong sign and a wrong or inconsistent sort. Integer.compare(a, b) returns the correct sign always.
2. When you need keys in sorted order, the nearest key (floorKey, ceilingKey), min/max key, or range views. It costs $O(\log n)$ per operation instead of $O(1)$ average.
3. path is one object that keeps changing. The result stores a reference, so every saved entry shows its final (often empty) state. Add `new ArrayList<>(path)` instead.

E. Long Answers

1. Model answer:
 - String is immutable; every modification creates a new object.
 - `s += c` in a loop copies the whole string each time: $O(n^2)$ total.
 - StringBuilder is a mutable, resizable char buffer; append is amortised $O(1)$.
 - StringBuilder offers reverse, insert, deleteCharAt, setCharAt, setLength.
 - Compare Strings with equals, never `==`.
2. Model answer:
 - ArrayList: dynamic array, get/set $O(1)$, add at end amortised $O(1)$, insert/remove in middle $O(n)$.
 - HashMap: key-value, get/put/containsKey $O(1)$ average; counting, indexing, caching.
 - HashSet: membership and duplicates, $O(1)$ average.
 - TreeMap/TreeSet: sorted, $O(\log n)$, floor/ceiling, first/last, ranges.
 - ArrayDeque: stack (push/pop) and queue (offer/poll), $O(1)$; BFS and monotonic stacks.

- PriorityQueue: heap, offer/poll $O(\log n)$, peek $O(1)$; top-k, Dijkstra, scheduling.
- LinkedList: $O(1)$ at ends, $O(n)$ by index; rarely better than ArrayDeque.

F. Design Challenge: Model Solution

(1) First Unique Character: `int[26]` counts, then first index with count 1; $O(n)$, $O(1)$. (2) Move Zeroes: copy non-zeros to index `w++`, fill the rest with 0; $O(n)$, $O(1)$. (3) Group Anagrams: `key = new String(sortedChars)`, then `computeIfAbsent(key, k → new ArrayList<>()).add(word)`; $O(n \cdot L \log L)$. (4) Queue using Stacks: push to "in"; on pop/peek, if "out" is empty, move all of in to out; amortised $O(1)$. (5) Top K Frequent: count with merge, min-heap by count, poll when size exceeds `k`; $O(n \log k)$ time, $O(n)$ space.

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 the Java collections, StringBuilder and comparators fluently, and avoid integer overflow.

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 the Java 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 (int or long)?

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 HashMap 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 Java: good names (`left` , `right` , `seen`), small helper methods, 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 <code>target - x</code> . A HashMap 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 {
    public int[] twoSum(int[] nums, int target) {
        Map<Integer, Integer> index0f = new HashMap<>();
        for (int i = 0; i < nums.length; i++) {
            int need = target - nums[i];
            if (index0f.containsKey(need)) {           // partner seen earlier
                return new int[] { index0f.get(need), i };
            }
            index0f.put(nums[i], i);                  // store AFTER the check
        }
        return new int[] { -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 . If values are up to 10^9 and you add two of them, the sum is 2×10^9 , which overflows a Java `int` (max about 2.147×10^9). If values are small (0 to 100), a counting array of size 101 can replace a HashMap 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 (+ HashMap)	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 (PriorityQueue)	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 HashMap." 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 HashSet or HashMap 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. Compilers catch syntax; 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

```
int findMax(int[] a) {
    int best = 0;                // suspicious start value
    for (int i = 0; i < a.length; i++) {
        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 `Integer.MIN_VALUE`. 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 the integer limits. 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 HashMap 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 Java Essentials for Interviews

Java is verbose, so you must know the API by heart. Fumbling with method names wastes minutes and confidence. This section is your cheat sheet.

1.6.1 Arrays

```
int[] a = new int[n];           // filled with 0
int[] b = {5, 2, 9};
Arrays.sort(b);                 // ascending, O(n log n)
Arrays.fill(a, -1);             // set all to -1
int[] copy = Arrays.copyOf(b, b.length);
int[] part = Arrays.copyOfRange(b, 1, 3); // indices 1..2, end exclusive
String s = Arrays.toString(b);  // "[2, 5, 9]" for debugging
int[][] grid = new int[rows][cols];
Integer[] boxed = {3, 1, 2};
Arrays.sort(boxed, (x, y) → y - x); // custom order needs objects, not int[]
```

1.6.2 Collections cheat table

Need	Java class	Key methods	Time
Dynamic array	ArrayList	add, get(i), set(i,x), remove(size-1), size	get/set O(1); add O(1) amortised; remove from middle O(n)
Key → value	HashMap	put, get, getOrDefault, containsKey, remove, merge, entrySet	O(1) average
Unique items	HashSet	add (returns false if present), contains, remove	O(1) average
Stack	ArrayDeque	push, pop, peek, isEmpty	O(1)
Queue	ArrayDeque	offer, poll, peek, isEmpty	O(1)
Deque (both ends)	ArrayDeque	offerFirst/offerLast, pollFirst/pollLast, peekFirst/peekLast	O(1)
Min-heap	PriorityQueue	offer, poll, peek, size	offer/poll O(log n), peek O(1)
Max-heap	PriorityQueue(Collections.reverseOrder())	same as above	O(log n)
Sorted map	TreeMap	floorKey, ceilingKey, firstKey, lastKey	O(log n)
Sorted set	TreeSet	floor, ceiling, first, last	O(log n)

```

Map<Character, Integer> freq = new HashMap<>();
for (char c : "banana".toCharArray()) {
    freq.merge(c, 1, Integer::sum);           // count in one line
}
int countA = freq.getOrDefault('a', 0);      // 3

for (Map.Entry<Character, Integer> e : freq.entrySet()) {
    System.out.println(e.getKey() + " " + e.getValue());
}

Map<String, List<String>> groups = new HashMap<>();
groups.computeIfAbsent("key", k → new ArrayList<>()).add("word");

Deque<Integer> stack = new ArrayDeque<>();
stack.push(1); stack.push(2);
int top = stack.pop();                       // 2

Deque<int[]> queue = new ArrayDeque<>();
queue.offer(new int[] {0, 0});               // BFS on grid cells
int[] cell = queue.poll();

PriorityQueue<Integer> minHeap = new PriorityQueue<>();
PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder());

```

⚠ COMMON MISTAKE

Using `Stack` and `LinkedList` out of habit. `Stack` is an old synchronised class; `ArrayDeque` is the recommended choice for both stacks and queues. Also note: `ArrayDeque` does not allow `null` elements.

1.6.3 Strings and StringBuilder

Java `String` is immutable. Every `s = s + c` inside a loop creates a new string and copies the old one, so building a string of length n this way costs $O(n^2)$. Use `StringBuilder`.

```
String s = "hello";
char c = s.charAt(1);           // 'e'
int len = s.length();          // method for String, field for arrays
String sub = s.substring(1, 4); // "ell", end exclusive
char[] chars = s.toCharArray();
String back = new String(chars);
boolean same = s.equals("hello"); // never use == for content

StringBuilder sb = new StringBuilder();
for (int i = 0; i < 3; i++) sb.append(i); // 0(1) amortised per append
sb.reverse();
sb.setCharAt(0, 'x');
sb.deleteCharAt(sb.length() - 1);
String result = sb.toString();

int digit = '7' - '0';           // 7
int idx = 'c' - 'a';             // 2, index into int[26]
char next = (char) ('a' + 3);    // 'd'
```

1.6.4 Comparators

```
int[][] intervals = {{5, 6}, {1, 3}, {2, 4}};
Arrays.sort(intervals, (x, y) → Integer.compare(x[0], y[0])); // by start

List<String> words = new ArrayList<>(List.of("pear", "fig", "apple"));
words.sort(Comparator.comparingInt(String::length)
    .thenComparing(Comparator.naturalOrder()));

// heap of [value, index] pairs, smallest value first
PriorityQueue<int[]> pq = new PriorityQueue<>((x, y) → Integer.compare(x[0], y[0]));
```

⚠ COMMON MISTAKE

Writing a comparator as $(x, y) \rightarrow x - y$. If $x = 2,000,000,000$ and $y = -2,000,000,000$, the subtraction overflows and the sign flips, giving a wrong order. Use `Integer.compare(x, y)`. Another classic: comparing two `Integer` objects with `=`. It works for values between -128 and 127 (cached) and fails silently above that. Use `.equals()` or unbox to `int`.

1.6.5 Integer overflow

Type	Range	Rough size
int	-2,147,483,648 to 2,147,483,647	about $\pm 2.1 \times 10^9$
long	about -9.22×10^{18} to 9.22×10^{18}	about $\pm 9.2 \times 10^{18}$

```
int a = 2_000_000_000, b = 2_000_000_000;
int wrong = a + b;                // -294967296 (silent overflow)
long right = (long) a + b;        // 4000000000, cast BEFORE adding
long alsoWrong = a + b;          // still overflows: int add happens first

int mid = lo + (hi - lo) / 2;     // safe midpoint, never (lo + hi) / 2
long product = (long) x * y;     // products of two ints need long
int MOD = 1_000_000_007;
hash = (hash * 31 + val) % MOD;  // hash is a long; keep numbers small in counting
problems
```

KEY POINT

If the problem says values up to 10^9 and you add or multiply them, use `long`. If it says "return the answer modulo $10^9 + 7$ ", take `% MOD` after every addition and multiplication.

DO YOU KNOW?

The "`(lo + hi) / 2`" overflow bug sat inside the JDK's own `Arrays.binarySearch` for about nine years. Joshua Bloch wrote about it on the Google Research blog in 2006 ("Nearly All Binary Searches and Mergesorts are Broken"). The fix was exactly `lo + (hi - lo) / 2` (or an unsigned shift).

AT WORK

Overflow is a real production bug class. Summing order amounts in paise across a large day of transactions can cross 2.1×10^9 quickly (that is only about ₹2.1 crore). Money and counters in backend code are usually `long` or `BigDecimal`, and reviewers will flag an `int` total.

INTERVIEW CORNER

Q: HashMap vs TreeMap: when would you pick each?

A: HashMap for $O(1)$ average lookups when order does not matter. TreeMap (a red-black tree) when you need sorted keys or "nearest key" queries like `floorKey` and `ceilingKey`, at $O(\log n)$ per operation.

Q: Why is string concatenation in a loop slow in Java?

A: Strings are immutable, so each `+` creates a new string and copies all previous characters, giving $O(n^2)$ total. `StringBuilder` appends into a growable buffer in $O(1)$ amortised.

✓ CHECK YOUR PROGRESS 1.6

1. Which Java class should you use for a stack, and why not `Stack` ?

Answer: `ArrayDeque` with `push/pop/peek`. `Stack` extends `Vector`, is synchronised (slower), and the Java docs themselves recommend `Deque` implementations instead.

2. What does `long x = a * b;` compute when `a` and `b` are ints near 10^5 each?

Answer: $a \times b \approx 10^{10}$, which overflows in int arithmetic before being widened to long, so `x` holds a wrong value. Write `(long) a * b`.

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 `ArrayList`, `HashMap`, `HashSet`, `ArrayDeque`, `PriorityQueue` and `TreeMap` APIs by heart.
- Use `StringBuilder` for building strings; compare strings and Integer objects with `equals`.
- Use `Integer.compare` in comparators, long for sums and products, and `lo + (hi - lo) / 2` for midpoints.

Key terms: problem-solving framework, clarifying question, brute force, BUD optimisation, constraint analysis, pattern recognition, dry run, edge case, `ArrayDeque`, `PriorityQueue`, `HashMap`, `TreeMap`, `StringBuilder`, comparator, integer overflow, 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 Java class is recommended for a stack?
 - Stack
 - ArrayDeque
 - Vector
 - TreeSet
- Which comparator is safe from overflow?
 - $(x, y) \rightarrow x - y$
 - $(x, y) \rightarrow y - x$
 - $(x, y) \rightarrow \text{Integer.compare}(x, y)$
 - $(x, y) \rightarrow x + y$
- A problem asks for "all permutations" of a list with $n \leq 8$. The pattern is:
 - Sliding window
 - Backtracking
 - Binary search
 - Prefix sum
- What does `Integer a = 1000, b = 1000; a == b` usually return?
 - true
 - false
 - compile error
 - NullPointerException

B. Fill in the Blanks

- The safe midpoint formula is $\text{mid} = \text{lo} + \underline{\hspace{2cm}}$.
- To make a max-heap of Integers, pass `_____` to the PriorityQueue constructor.
- Building a string with `+` inside a loop costs `_____` time in total.

C. True or False

- A dry run should trace the idea, not the exact code.
- HashMap get is $O(1)$ on average.
- `long x = a + b;` with int a and b never overflows.

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 count array 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 Java 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) ArrayDeque is fast and unsynchronised; Stack is legacy.
3. (c) Integer.compare never subtracts, so it cannot overflow.
4. (b) "All" plus tiny n means backtracking, $O(n \cdot n!)$.
5. (b) Values outside -128 to 127 are not cached, so `==` compares two different objects.

B. Fill in the Blanks

1. $(hi - lo) / 2$
2. `Collections.reverseOrder()`
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, but average is $O(1)$ (Java 8+ also treeifies large buckets, giving $O(\log n)$ worst case per bucket).
3. False. $a + b$ is computed in `int` first, then widened. Cast one operand: `(long) a + b`.

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, and use a `HashSet`; if `add` returns false, a duplicate exists. Time $O(n)$, space $O(n)$. 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 `Integer.MIN_VALUE`) and scan. 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.
 - 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 overflow and for counting-array 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 fills `int[26]` counts; second pass returns the first i with count 1.
- Code: `count[s.charAt(i) - 'a']++`, then a second loop.
- Test: "aabb" returns -1; single "z" returns 0.
- Analyse: $O(n)$ time, $O(1)$ space (26 is constant).

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 (overflow). (4) Java pitfalls: Integer == comparison, x - y comparators, int sums that need long. Add one line at the bottom: "Brute force out loud, then optimise, then ask before coding."