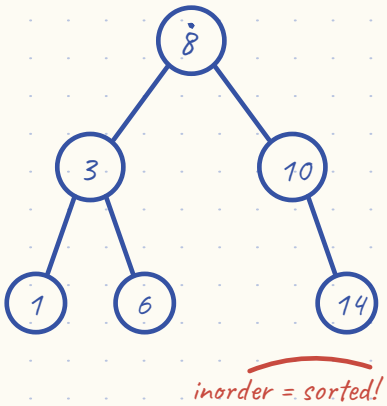

Interview prep notebook · Java

DSA Notes

Data Structures & Algorithms from basic to advanced — simple words, real-life examples, cost tables, Java code, and a practice list for every topic.

34 topics · 8 levels · 240 practice problems · ≈91 study sessions

name / started on

Contents

Tick the box when a topic is done (3–5 problems solved).

Level 4 — Backtracking & Greedy

Level 5 — Graphs

Level 6 — Dynamic Programming

Level 7 — Advanced & Design

Quick revision

How to study

One 45-minute session

1. **Read (10 min)** — "In simple words", the sticky note, and the key ideas.
2. **Write the code (5 min)** — close the notes, write the Java code from memory, then compare.
3. **Solve one problem (25 min)** — Easy first, then Medium. Stuck for 20 min? Read a solution, re-solve it tomorrow without help.
4. **Note it (5 min)** — one line: the pattern you used + the mistake you made.

Go **top to bottom**, one topic at a time. Each level builds on the one before.

Difficulty marks

(E) Easy (M) Medium (H) Hard

All problem names match LeetCode titles.

Solving in the interview

1. **Understand** — repeat the problem; ask about input size, ranges, duplicates, empty input.
2. **Examples** — walk through the example, add one edge case of your own.
3. **Brute force** — say the simple solution and its complexity.
4. **Optimize** — find the bottleneck, name the pattern.
5. **Code** — clear names, small helper methods, talk while typing.
6. **Test** — dry-run on the example, then edge cases.
7. **Complexity** — finish with time and space, without being asked.

Level 0

Foundations

The language of complexity, the Java tools you'll use daily, and the recursion and math every later topic leans on.

In this level: Time & Space Complexity (Big-O) · Java Toolkit for DSA · Recursion · Math Basics for Coding · Bit Manipulation

Time & Space Complexity (Big-O)

Level 0 · Foundations · ≈ 2 sessions

In simple words

*Big-O tells you **how the work grows when the input grows**. We don't measure seconds (they change from laptop to laptop). We count **steps** as the input size `n` becomes very large.*

*Two simple rules: **drop constants** (O(2n) is just O(n)) and **keep only the biggest term** (O(n² + n) is O(n²)). If there are two different inputs, use two letters: O(n · m).*

> **Think of it like...**
>
> *Finding a name in a phone book. Turning page by page is O(n). Opening the middle, then the middle of the correct half, and so on, is O(log n) — a 1,000-page book needs only about 10 openings.*

Key ideas

- ***O(1)** constant — array index access, HashMap get. Same work for 10 or 10 million items.*
- ***O(log n)** — you cut the problem in half every step (binary search, balanced tree).*
- ***O(n)** — one pass over the data (one loop).*
- ***O(n log n)** — good sorting (merge sort, `Arrays.sort` on objects).*
- ***O(n²)** — a loop inside a loop over the same data.*
- ***O(2ⁿ)** — trying every subset. **O(n!)** — trying every ordering (permutation).*
- ***Space complexity** = extra memory you create: new arrays, maps, and the **recursion call stack** (depth counts!).*
- ***Amortized** cost = average over many operations. `ArrayList.add` is O(1) amortized even though a resize sometimes costs O(n).*

✱ Best / average / worst case: interviews usually mean **worst case** unless you say otherwise (e.g. HashMap is O(1) average).

Cost sheet

Input size n (from constraints)	Complexity that will pass (~10^8 steps/sec)
$n \le 10$	$O(n!)$ — permutations
$n \le 20$	$O(2^n)$ — subsets / bitmask
$n \le 500$	$O(n^3)$
$n \le 10^4$	$O(n^2)$
$n \le 10^6$	$O(n \log n)$ or $O(n)$
$n \le 10^9$ or more	$O(\log n)$ or $O(1)$ — math / binary search

Java code

```
// O(1): one step no matter how big the array is
int first = arr[0];
```

```
// O(n): one loop
for (int x : arr) sum += x;
```

```
// O(n^2): loop inside a loop
for (int i = 0; i < n; i++)
   for (int j = i + 1; j < n; j++)
      if (arr[i] + arr[j] == target) return true;
```

```
// O(log n): the range halves every step
while (n > 1) { n = n / 2; steps++; }
```

```
// O(n) SPACE: extra array of size n
int[] copy = new int[n];
```

```
// Recursion: depth n  => O(n) stack space even with no arrays
int sumTo(int n) { return n == 0 ? 0 : n + sumTo(n - 1); }
```

Spot it when

→ The constraints line (like $1 \le n \le 10^5$) quietly tells you the expected complexity — use the table above.

→ $n \le 10^5$ usually means $O(n^2)$ is too slow; aim for $O(n \log n)$ or $O(n)$.

★ **Interview tip:** Finish every answer by saying the time **and** space complexity out loud — before the interviewer asks. It signals seniority.

Java Toolkit for DSA

Level 0 · Foundations · ≈ 1 session

In simple words

Half of solving a problem quickly is **knowing which Java collection to reach for**. You already use many of these at work; here is how they map to DSA ideas and what each operation costs.

> **Think of it like...**
>
> Like a mechanic's toolbox: the job is easy once you pick the right spanner.

Key ideas

- Use `ArrayDeque` for stacks — the old `Stack` class is synchronized and legacy.
- Never compare `Integer` objects with `==`. It works for −128..127 (cache) and silently fails above. Use `.equals()` or unbox to `int`.
- Sums and products overflow `int` (max ≈ 2.1×10^9). Use `long`, and write `(long) a * b`.
- Comparator trap: `(a, b) -> a - b` can overflow. Prefer `Integer.compare(a, b)`.
- `Arrays.sort(int[])` uses dual-pivot quicksort (not stable); `Arrays.sort(Object[])` and `Collections.sort` use TimSort (stable).
- String concatenation inside a loop is $O(n^2)$ because strings are immutable — use `StringBuilder`.
- Handy map methods: `getOrDefault`, `merge`, `computeIfAbsent`, `putIfAbsent`.

Cost sheet

Class	DSA idea	Main costs
ArrayList	Dynamic array	get O(1) · add at end O(1)* · insert/remove middle O(n)
HashMap / HashSet	Hash table	put / get / contains O(1) average
LinkedHashMap	Hash table + insertion order	O(1); can evict eldest → LRU cache
TreeMap / TreeSet	Balanced BST (Red-Black)	O(log n) · floorKey, ceilingKey, firstKey
ArrayDeque	Stack and Queue	push / pop / offer / poll O(1)
PriorityQueue	Binary heap (min by default)	offer / poll O(log n) · peek O(1)
StringBuilder	Mutable string	append O(1)* · toString O(n)
int[] / Arrays	Fixed array	Arrays.sort O(n log n) · Arrays.fill O(n)

Java code

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

```java
// Frequency count in one line
Map<Character, Integer> freq = new HashMap<>();
for (char c : s.toCharArray()) freq.merge(c, 1, Integer::sum);
```

```java
// Map of lists (adjacency list, grouping)
Map<String, List<String>> groups = new HashMap<>();
groups.computeIfAbsent(key, k -> new ArrayList<>()).add(word);
```

```java
// Stack and queue
Deque<Integer> stack = new ArrayDeque<>();
stack.push(1); stack.pop(); stack.peek();
Queue<Integer> queue = new ArrayDeque<>();
queue.offer(1); queue.poll(); queue.peek();
```

```java
// Min-heap and max-heap
PriorityQueue<Integer> minHeap = new PriorityQueue<>();
PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder());
// Heap of int[] sorted by second value
PriorityQueue<int[]> pq = new PriorityQueue<>((a, b) -> Integer.compare(a[1], b[1]));
```

```java
// Sorted map: nearest keys
TreeMap<Integer, String> tm = new TreeMap<>();
Integer floor = tm.floorKey(10);   // largest key <= 10, or null
Integer ceil  = tm.ceilingKey(10); // smallest key >= 10, or null
```

```java
// Sort 2D array by start time
Arrays.sort(intervals, (a, b) -> Integer.compare(a[0], b[0]));
```

```java
// Safe math
long product = (long) a * b;
int mid = lo + (hi - lo) / 2;     // avoids overflow of (lo + hi)
```

Spot it when

- Need order + fast lookup → `TreeMap`.
- Need 'most recent' → `ArrayDeque` as stack.
- Need 'smallest/largest so far' repeatedly → `PriorityQueue`.

★ **Interview tip:** Say why you picked a structure: "I'll use a TreeMap because I need the nearest smaller key in O(log n)." Interviewers score the reasoning, not only the code.

Recursion

Level 0 · Foundations · ≈ 2 sessions

In simple words

Recursion is a function that **calls itself on a smaller version of the same problem**. Every recursive function has two parts:

1. **Base case** — the smallest input where you already know the answer (stop here).
2. **Recursive case** — break the problem into a smaller piece, call yourself, and combine the result.

The trick is to **trust** that the smaller call works, just like you trust a library method.

Think of it like...

You are in a cinema queue and want to know your position. You ask the person in front, who asks the person in front of them... The first person says "1" (base case). Each person adds 1 and passes it back.

Key ideas

- Each call is stored on the **call stack**. Too deep (roughly 10,000+ in default Java) → `StackOverflowError`.
- Draw the **recursion tree** to find complexity: (number of calls) × (work per call).
- Naive Fibonacci makes ~2^n calls because it solves the same subproblem again and again → fix with **memoization** (store answers). This is the doorway to Dynamic Programming.
- Java does not optimize tail recursion; convert very deep recursion to a loop or an explicit stack.
- Recursion is the backbone of trees, graphs (DFS), backtracking, divide & conquer, and DP.

Cost sheet

Example	Time	Space (stack)
factorial(n)	O(n)	O(n)
naive fib(n)	$O(2^n)$	O(n)
memo fib(n)	O(n)	O(n)
fast power(x, n)	O(log n)	O(log n)

Java code

```java
// Factorial
long fact(int n) {
    if (n <= 1) return 1;       // base case
    return n * fact(n - 1);     // recursive case
}
```

```java
// Fibonacci with memoization: O(n) instead of O(2^n)
long[] memo = new long[100];
long fib(int n) {
  if (n <= 1) return n;
  if (memo[n] != 0) return memo[n];
  return memo[n] = fib(n - 1) + fib(n - 2);
}
```

```java
// Fast power: x^n in O(log n)
double myPow(double x, long n) {
  if (n == 0) return 1;
  if (n < 0) return 1 / myPow(x, -n);
  double half = myPow(x, n / 2);
  return (n % 2 == 0) ? half * half : half * half * x;
}
```

Spot it when

- → *The problem is defined in terms of itself (tree, nested structure, 'for each choice...').*
- → *You can say: 'if I knew the answer for n-1, I could get n'.*

Practice list

- ☐ *Fibonacci Number* (E)
- ☐ *Reverse Linked List* (E)
- ☐ *Merge Two Sorted Lists* (E)
- ☐ *Pow(x, n)* (M)
- ☐ *K-th Symbol in Grammar* (M)

★ **Interview tip:** *On the whiteboard, draw the recursion tree for n = 3 or 4. It makes the complexity obvious and shows the interviewer how you think.*

Math Basics for Coding

Level 0 · Foundations · ≈ 1 session

In simple words

A handful of math tricks show up again and again: ***GCD***, ***prime numbers***, ***modulo arithmetic*** *and* ***fast power***. *The focus here is on writing them cleanly and safely in Java.*

Think of it like...

These are the multiplication tables of coding interviews — small facts that make bigger problems quick.

Key ideas

- **GCD (Euclid)**: gcd(a, b) = gcd(b, a % b), stop when b = 0. O(log min(a,b)). LCM = a / gcd(a,b) × b.
- **Is n prime?** Check divisors only up to √n → O(√n).
- **Sieve of Eratosthenes**: all primes up to n in O(n log log n).
- **Modulo 10^9+7**: large answers are asked 'mod M. Apply mod after every add/multiply, use `long` for the multiply.
- Negative mod in Java: `-7 % 3 == -1`. Use `Math.floorMod(a, m)` or `((a % m) + m) % m`.
- **Digits**: `n % 10` gives the last digit, `n / 10` removes it.
- **nCr** with Pascal's triangle: C(n, r) = C(n−1, r−1) + C(n−1, r).

Cost sheet

Task	Cost
gcd(a, b)	O(log min(a, b))
isPrime(n)	O(√n)
Sieve up to n	O(n log log n)
modPow(b, e, m)	O(log e)

Java code

```
int gcd(int a, int b) { return b == 0 ? a : gcd(b, a % b); }
```

```
boolean[] sieve(int n) {          // true = composite
   boolean[] notPrime = new boolean[n + 1];
   for (int i = 2; (long) i * i <= n; i++)
      if (!notPrime[i])
         for (int j = i * i; j <= n; j += i) notPrime[j] = true;
   return notPrime;
}
```

```
static final int MOD = 1_000_000_007;
long modPow(long base, long exp, long mod) {
   long result = 1; base %= mod;
   while (exp > 0) {
      if ((exp & 1) == 1) result = result * base % mod;
      base = base * base % mod;
      exp >>= 1;
   }
   return result;
}
```

Spot it when

→ 'Return the answer modulo 10^9+7' → the answer is huge; keep taking mod.

→ 'Count primes', 'divisible by', 'digits of a number'.

Practice list

- [ ] Happy Number (E)
- [ ] Excel Sheet Column Number (E)
- [ ] Count Primes (M)
- [ ] Reverse Integer (M)
- [ ] Pow(x, n) (M)

★ **Interview tip:** In "reverse integer" style problems, check for overflow *before* multiplying by 10. Interviewers watch for that edge case.

Bit Manipulation

Level 0 · Foundations · ≈ 2 sessions

In simple words

Computers store numbers as 0s and 1s. Bit operators let you work on those bits directly — very fast and often O(1) extra space.

`&` AND · `|` OR · `^` XOR (1 when bits differ) · `~` NOT · `<<` left shift (×2) · `>>` right shift (÷2, keeps sign) · `>>>` unsigned right shift (Java-specific).

Think of it like...

A row of light switches. Each operator is a way of flipping, checking, or copying switches.

Key ideas

- `(x & 1) == 1` → x is odd.
- `x & (x - 1)` removes the lowest 1-bit. `x > 0 && (x & (x-1)) == 0` → x is a power of two.
- `x ^ x = 0` and `x ^ 0 = x` → XOR of all numbers cancels the pairs (Single Number).
- Check bit i: `(x >> i) & 1` · Set: `x | (1 << i)` · Clear: `x & ~(1 << i)` · Toggle: `x ^ (1 << i)`.
- `x & -x` isolates the lowest set bit (used in Fenwick trees).
- `Integer.bitCount(x)` counts 1-bits. A mask from 0 to 2^n-1 can represent every subset of n items.

Cost sheet

Operation	Expression
Is odd	`(x & 1) == 1`
Power of two	`x > 0 && (x & (x - 1)) == 0`
Get bit i	`(x >> i) & 1`
Set bit i	`x | (1 << i)`
Clear bit i	`x & ~(1 << i)`
Lowest set bit	`x & -x`

Java code

```java
// Single Number: every number appears twice except one
int singleNumber(int[] nums) {
    int ans = 0;
    for (int x : nums) ans ^= x;   // pairs cancel out
    return ans;
}
```

```java
// Count 1-bits (Brian Kernighan): runs once per set bit
int countOnes(int x) {
    int count = 0;
    while (x != 0) { x &= (x - 1); count++; }
    return count;
}
```

```java
// All subsets using a bitmask
List<List<Integer>> subsets(int[] nums) {
    int n = nums.length;
    List<List<Integer>> res = new ArrayList<>();
    for (int mask = 0; mask < (1 << n); mask++) {
        List<Integer> cur = new ArrayList<>();
        for (int i = 0; i < n; i++)
            if (((mask >> i) & 1) == 1) cur.add(nums[i]);
        res.add(cur);
    }
    return res;
}
```

Spot it when

- → *'Appears once / twice', 'without extra space'.*
- → *$n \leq 20$ and you need 'all subsets' → bitmask.*
- → *'Without using + or −' → XOR and carry.*

Practice list

- [ ] Single Number (E)
- [ ] Number of 1 Bits (E)
- [ ] Counting Bits (E)
- [ ] Missing Number (E)
- [ ] Reverse Bits (E)
- [ ] Subsets (M)
- [ ] Sum of Two Integers (M)

★ **Interview tip:** Put parentheses around bit expressions. In Java `==` binds tighter than `&`, so `x & 1 == 1` does not compile.

Level 1

Basic Data Structures

The building blocks: arrays, strings, hash tables, linked lists, stacks and queues.

In this level: Arrays · Strings · Hashing (HashMap & HashSet) · Linked List · Stack · Queue & Deque

Arrays

Level 1 · Basic Data Structures · ≈ 3 sessions

In simple words

An array is a **row of boxes of the same type**, each with an index starting at 0. Getting any box is instant because the computer calculates its address: *start + index × size*.

The cost: inserting or deleting in the middle means **shifting** everything after it. 2D arrays (`matrix[row][col]`) are just arrays of arrays.

> **Think of it like...**
>
> Numbered lockers in a corridor. You walk straight to locker 42 — but adding a new locker between 5 and 6 means moving every locker after it.

Key ideas

- **Kadane's algorithm**: maximum subarray sum in O(n). Keep a running sum; if it goes negative, start fresh.
- **Reverse trick**: rotate an array by k = reverse all, reverse first k, reverse the rest. O(n) time, O(1) space.
- **Dutch National Flag**: sort 0s, 1s, 2s in one pass with three pointers.
- **In-place marking**: use the array itself (e.g. negate values, or place value v at index v−1) to save space.
- **Prefix / suffix products**: 'product except self' without division.
- Matrix tricks: rotate 90° = transpose + reverse each row; spiral traversal with four boundaries.
- Always check **empty array**, **single element**, **all negatives**, and **index out of bounds**.

Cost sheet

Operation	Time
Access arr[i]	O(1)
Search (unsorted)	O(n)
Search (sorted, binary search)	O(log n)
Insert / delete at end (ArrayList)	O(1) amortized
Insert / delete in middle	O(n)
Sort	O(n log n)

Java code

```java
// Kadane: largest sum of a contiguous subarray
int maxSubArray(int[] nums) {
    int best = nums[0], cur = 0;
    for (int x : nums) {
        cur = Math.max(x, cur + x);  // extend or restart
        best = Math.max(best, cur);
    }
    return best;
}
```

```java
// Rotate right by k using three reverses
void rotate(int[] a, int k) {
    k %= a.length;
    reverse(a, 0, a.length - 1);
    reverse(a, 0, k - 1);
    reverse(a, k, a.length - 1);
}
void reverse(int[] a, int i, int j) {
    while (i < j) { int t = a[i]; a[i++] = a[j]; a[j--] = t; }
}
```

```java
// Best time to buy and sell stock (one transaction)
int maxProfit(int[] prices) {
    int minPrice = Integer.MAX_VALUE, profit = 0;
    for (int p : prices) {
        minPrice = Math.min(minPrice, p);
        profit = Math.max(profit, p - minPrice);
    }
    return profit;
}
```

Spot it when

→ Almost every problem starts with an array — decide next which technique (hashing, two pointers, window, prefix sum) fits.

Practice list

- [ ] Two Sum (E)
- [ ] Best Time to Buy and Sell Stock (E)
- [ ] Maximum Subarray (M)
- [ ] Product of Array Except Self (M)
- [ ] Rotate Array (M)
- [ ] Sort Colors (M)
- [ ] Spiral Matrix (M)
- [ ] Rotate Image (M)
- [ ] Set Matrix Zeroes (M)
- [ ] First Missing Positive (H)

★ **Interview tip:** Before coding, ask: Is it sorted? Can there be duplicates or negatives? Can I modify the input? The answers often decide the technique.

Strings

Level 1 · Basic Data Structures · ≈ 2 sessions

In simple words

A string is a sequence of characters. In Java, `String` is ***immutable*** — every "change" creates a new object. For building strings use `StringBuilder`; for checking characters use `char[]` or `charAt`.

Think of it like...

A printed sentence. To change one word you reprint the whole line — unless you use a whiteboard (StringBuilder).

Key ideas

- Count letters with `int[26]` and `c - 'a'` — faster and simpler than a HashMap.
- ***Palindrome***: two pointers from both ends moving inward.
- ***Anagram***: same letter counts. Group anagrams by their sorted form or their count signature.
- Useful helpers: `Character.isLetterOrDigit`, `Character.toLowerCase`, `String.join`, `s.split("\\s+")`.
- Expand-around-center finds the longest palindromic substring in $O(n^2)$ with $O(1)$ space.
- Many string problems are really ***sliding window***, ***hashing***, or ***DP*** problems in disguise.

Cost sheet

Operation	Time
charAt(i)	O(1)
length()	O(1)
substring(i, j)	O(j − i) — copies
equals / compareTo	O(n)
s + t inside a loop	O(n²) overall — avoid
StringBuilder.append	O(1) amortized

Java code

```
// Valid palindrome, ignoring non-alphanumeric characters
boolean isPalindrome(String s) {
  int i = 0, j = s.length() - 1;
  while (i < j) {
    while (i < j && !Character.isLetterOrDigit(s.charAt(i))) i++;
    while (i < j && !Character.isLetterOrDigit(s.charAt(j))) j--;
    if (Character.toLowerCase(s.charAt(i)) != Character.toLowerCase(s.charAt(j)))
      return false;
    i++; j--;
  }
  return true;
}
```

```
// Anagram check with a count array
boolean isAnagram(String s, String t) {
  if (s.length() != t.length()) return false;
  int[] count = new int[26];
  for (int i = 0; i < s.length(); i++) {
    count[s.charAt(i) - 'a']++;
    count[t.charAt(i) - 'a']--;
  }
  for (int c : count) if (c != 0) return false;
  return true;
}
```

```java
// Longest palindromic substring: expand around each center
String longestPalindrome(String s) {
  int start = 0, end = 0;
  for (int c = 0; c < s.length(); c++) {
    int len = Math.max(expand(s, c, c), expand(s, c, c + 1));
    if (len > end - start) { start = c - (len - 1) / 2; end = c + len / 2; }
  }
  return s.substring(start, end + 1);
}
int expand(String s, int l, int r) {
  while (l >= 0 && r < s.length() && s.charAt(l) == s.charAt(r)) { l--; r++; }
  return r - l - 1;
}
```

Spot it when

→ *'Anagram', 'palindrome', 'substring', 'character frequency'.*

Practice list

- ☐ *Valid Anagram* (E)
- ☐ *Valid Palindrome* (E)
- ☐ *Longest Common Prefix* (E)
- ☐ *Group Anagrams* (M)
- ☐ *Longest Palindromic Substring* (M)
- ☐ *String to Integer (atoi)* (M)
- ☐ *Reverse Words in a String* (M)
- ☐ *Encode and Decode Strings* (M)

★ **Interview tip:** *Clarify the character set first: only lowercase English letters? Upper case? Unicode? It decides between* `int[26]`, `int[128]` *and a HashMap.*

Hashing (HashMap & HashSet)

Level 1 · Basic Data Structures · ≈ 2 sessions

In simple words

A hash table stores ***key → value*** *pairs and finds any key in* ***O(1) on average****. A hash function turns the key into a number, and that number picks a bucket.*

`HashSet` *is the same idea with keys only — perfect for "have I seen this before?"*

Think of it like...

A library catalog: instead of scanning every shelf, the catalog tells you exactly which shelf the book is on.

Key ideas

- **How Java's HashMap works** (a favorite interview question): array of buckets → `hashCode()` picks a bucket → `equals()` finds the exact key inside.
- Collisions are chained in a linked list; since Java 8 a bucket with more than 8 entries turns into a **red-black tree**.
- Load factor 0.75: when 75% full, the table doubles and all entries are rehashed.
- If you override `equals`, you must override `hashCode` — otherwise lookups fail.
- Keys should be immutable. A mutated key can land in the wrong bucket and 'disappear'.
- `HashMap` has no order · `LinkedHashMap` keeps insertion order · `TreeMap` keeps sorted order (O(log n)).
- Classic patterns: **frequency count**, **complement lookup** (Two Sum), **prefix sum + map**, **grouping by a signature**.

Cost sheet

Operation	Average	Worst
put / get / remove	O(1)	O(n) — O(log n) per bucket since Java 8
containsKey	O(1)	O(n)
Iterate all	O(n + capacity)	—

Java code

```
// Two Sum: find the complement in O(1)
int[] twoSum(int[] nums, int target) {
  Map<Integer, Integer> seen = new HashMap<>();  // value -> index
  for (int i = 0; i < nums.length; i++) {
    int need = target - nums[i];
    if (seen.containsKey(need)) return new int[]{seen.get(need), i};
    seen.put(nums[i], i);
  }
  return new int[0];
}
```

```
// Group anagrams by a sorted-letters key
List<List<String>> groupAnagrams(String[] words) {
  Map<String, List<String>> map = new HashMap<>();
  for (String w : words) {
    char[] c = w.toCharArray();
    Arrays.sort(c);
    map.computeIfAbsent(new String(c), k -> new ArrayList<>()).add(w);
  }
  return new ArrayList<>(map.values());
}
```

```java
// Longest consecutive sequence in O(n)
int longestConsecutive(int[] nums) {
    Set<Integer> set = new HashSet<>();
    for (int x : nums) set.add(x);
    int best = 0;
    for (int x : set) {
        if (set.contains(x - 1)) continue;        // only start at a run's beginning
        int len = 1;
        while (set.contains(x + len)) len++;
        best = Math.max(best, len);
    }
    return best;
}
```

Spot it when

- → *'Find a pair / duplicate / count occurrences'.*
- → *You want to turn an O(n²) search into O(n).*
- → *'Group items that are the same in some way'.*

Practice list

- ☐ *Two Sum* (E)
- ☐ *Contains Duplicate* (E)
- ☐ *Isomorphic Strings* (E)
- ☐ *Group Anagrams* (M)
- ☐ *Top K Frequent Elements* (M)
- ☐ *Longest Consecutive Sequence* (M)
- ☐ *Subarray Sum Equals K* (M)

★ **Interview tip:** *For Java roles, be ready to explain HashMap internals (buckets, hashCode/equals, treeification, resize) and why* `ConcurrentHashMap` *is used in multi-threaded services.*

Linked List

Level 1 · Basic Data Structures · ≈ 3 sessions

In simple words

*A linked list is a **chain of nodes**. Each node holds a value and a pointer to the next node. Nodes can live anywhere in memory, so there is **no index access** — you walk from the head.*

*Types: **singly** (next only), **doubly** (next and prev), **circular** (last points back to first).*

Think of it like...

A treasure hunt: each clue tells you where the next clue is hidden. To reach clue 10 you must follow clues 1 to 9.

Key ideas

- ***Dummy (sentinel) node***: put a fake node before the head so you never special-case 'the head changed'.
- ***Fast & slow pointers***: slow moves 1, fast moves 2. When fast ends, slow is in the ***middle***. If they ever meet, there is a ***cycle*** (Floyd's algorithm).
- ***Reverse*** with three pointers: prev, cur, next. Learn it until you can write it with your eyes closed.
- ***Nth from end***: move one pointer n steps ahead, then move both together.
- Many hard problems = reverse + find middle + merge, combined.
- Always ***draw boxes and arrows*** before changing pointers.

Cost sheet

Operation	Time
Access k-th node	O(n)
Search	O(n)
Insert / delete at head	O(1)
Insert / delete after a known node	O(1)
Insert at tail (with tail pointer)	O(1)

Java code

```
class ListNode {
  int val; ListNode next;
  ListNode(int v) { val = v; }
}
```

```
// Reverse a list iteratively
ListNode reverse(ListNode head) {
  ListNode prev = null, cur = head;
  while (cur != null) {
    ListNode next = cur.next;  // save
    cur.next = prev;           // flip
    prev = cur;                // move prev
    cur = next;                // move cur
  }
  return prev;
}
```

```
// Middle node (second middle for even length)
ListNode middle(ListNode head) {
  ListNode slow = head, fast = head;
  while (fast != null && fast.next != null) {
    slow = slow.next;
    fast = fast.next.next;
  }
  return slow;
}
```

```java
// Cycle detection (Floyd)
boolean hasCycle(ListNode head) {
    ListNode slow = head, fast = head;
    while (fast != null && fast.next != null) {
        slow = slow.next; fast = fast.next.next;
        if (slow == fast) return true;
    }
    return false;
}
```

```java
// Merge two sorted lists using a dummy node
ListNode merge(ListNode a, ListNode b) {
    ListNode dummy = new ListNode(0), tail = dummy;
    while (a != null && b != null) {
        if (a.val <= b.val) { tail.next = a; a = a.next; }
        else                { tail.next = b; b = b.next; }
        tail = tail.next;
    }
    tail.next = (a != null) ? a : b;
    return dummy.next;
}
```

Spot it when

- Input is a `ListNode`.
- 'In place', 'O(1) extra space', 'cycle', 'middle', 'k-th from end.

Practice list

- [ ] Reverse Linked List (E)
- [ ] Merge Two Sorted Lists (E)
- [ ] Linked List Cycle (E)
- [ ] Middle of the Linked List (E)
- [ ] Remove Nth Node From End of List (M)
- [ ] Reorder List (M)
- [ ] Add Two Numbers (M)
- [ ] Copy List with Random Pointer (M)
- [ ] LRU Cache (M)
- [ ] Merge k Sorted Lists (H)
- [ ] Reverse Nodes in k-Group (H)

★ **Interview tip:** Use a dummy head whenever the first node might be removed or replaced. It removes a whole class of null-pointer bugs.

Stack

Level 1 · Basic Data Structures · ≈ 2 sessions

In simple words

A stack is ***Last In, First Out (LIFO)***. You can only touch the top: `push` adds, `pop` removes, `peek` looks.

In Java: `Deque<Integer> stack = new ArrayDeque<>();`

> **Think of it like...**
>
> A pile of plates. You add a plate on top and you take the top plate first.

Key ideas

- Real uses: undo/redo, browser back button, the function call stack, expression parsing.
- **Matching pairs**: push opening brackets, pop when a closing bracket arrives.
- **Min stack**: store (value, minSoFar) together so getMin is O(1).
- Any recursion can be rewritten with an explicit stack (useful for very deep trees/graphs).
- **Monotonic stack** is a powerful special case — see Level 2.

Cost sheet

Operation	Time
push	O(1)
pop	O(1)
peek	O(1)
search	O(n)

Java code

```
// Valid Parentheses
boolean isValid(String s) {
    Deque<Character> st = new ArrayDeque<>();
    for (char c : s.toCharArray()) {
        if (c == '(') st.push(')');
        else if (c == '[') st.push(']');
        else if (c == '{') st.push('}');
        else if (st.isEmpty() || st.pop() != c) return false;
    }
    return st.isEmpty();
}
```

```
// Min Stack: every entry remembers the minimum below it
class MinStack {
    private final Deque<int[]> st = new ArrayDeque<>();   // {value, min}
    void push(int x) {
        int min = st.isEmpty() ? x : Math.min(x, st.peek()[1]);
        st.push(new int[]{x, min});
    }
    void pop()    { st.pop(); }
    int top()     { return st.peek()[0]; }
    int getMin()  { return st.peek()[1]; }
}
```

```java
// Evaluate Reverse Polish Notation: ["2","1","+","3","*"] -> 9
int evalRPN(String[] tokens) {
    Deque<Integer> st = new ArrayDeque<>();
    for (String t : tokens) {
        switch (t) {
            case "+" -> st.push(st.pop() + st.pop());
            case "*" -> st.push(st.pop() * st.pop());
            case "-" -> { int b = st.pop(), a = st.pop(); st.push(a - b); }
            case "/" -> { int b = st.pop(), a = st.pop(); st.push(a / b); }
            default  -> st.push(Integer.parseInt(t));
        }
    }
    return st.pop();
}
```

Spot it when

- 'Most recent unmatched thing' — brackets, tags, undo.
- Nested structures like `3[a2[c]]`.
- 'Next greater / smaller element'.

Practice list

- [ ] Valid Parentheses (E)
- [ ] Min Stack (M)
- [ ] Evaluate Reverse Polish Notation (M)
- [ ] Decode String (M)
- [ ] Asteroid Collision (M)
- [ ] Basic Calculator II (M)
- [ ] Largest Rectangle in Histogram (H)

★ **Interview tip:** When you see "most recent" or "undo the last", say "this smells like a stack" out loud — pattern naming earns points.

Queue & Deque

Level 1 · Basic Data Structures · ≈ 1 session

In simple words

A queue is **First In, First Out (FIFO)**: add at the back (`offer`), remove from the front (`poll`). A **deque** (double-ended queue) lets you add and remove at **both** ends.

In Java: `Queue<Integer> q = new ArrayDeque<>();` and `Deque<Integer> dq = new ArrayDeque<>();`

Think of it like...

The line at a ticket counter: whoever came first is served first. A deque is a line where people can also join or leave from the front.

Key ideas

- Queues power **BFS** (level-by-level traversal) in trees and graphs.
- **Circular queue** on a fixed array: move indices with `(i + 1) % capacity`.
- **Queue from two stacks**: push into 'in'; when 'out' is empty, pour everything from 'in' to 'out'. Amortized O(1).
- Deque powers the **sliding window maximum** (monotonic deque).
- Work link: Kafka topics are durable, distributed queues — a good bridge if the interviewer asks where you've used queues.

Cost sheet

Operation	Time
offer / add (back)	O(1)
poll / remove (front)	O(1)
peek	O(1)
offerFirst / pollLast (deque)	O(1)

Java code

```java
// Queue using two stacks
class MyQueue {
  private final Deque<Integer> in = new ArrayDeque<>(), out = new ArrayDeque<>();
  void push(int x) { in.push(x); }
  int pop()  { peek(); return out.pop(); }
  int peek() {
    if (out.isEmpty()) while (!in.isEmpty()) out.push(in.pop());
    return out.peek();
  }
  boolean empty() { return in.isEmpty() && out.isEmpty(); }
}
```

```java
// Circular queue on a fixed array
class CircularQueue {
  private final int[] data; private int head = 0, size = 0;
  CircularQueue(int k) { data = new int[k]; }
  boolean enQueue(int v) {
    if (size == data.length) return false;
    data[(head + size) % data.length] = v; size++;
    return true;
  }
  boolean deQueue() {
    if (size == 0) return false;
    head = (head + 1) % data.length; size--;
    return true;
  }
  int front() { return size == 0 ? -1 : data[head]; }
  int rear()  { return size == 0 ? -1 : data[(head + size - 1) % data.length]; }
}
```

Spot it when

- → 'Process in order of arrival', 'level by level', 'minimum steps' (BFS).
- → 'Last k seconds / recent calls' — a queue drops old items from the front.

Practice list

- ☐ Implement Queue using Stacks (E)
- ☐ Number of Recent Calls (E)
- ☐ Moving Average from Data Stream (E)
- ☐ Design Circular Queue (M)
- ☐ Rotting Oranges (M)
- ☐ Sliding Window Maximum (H)

★ **Interview tip:** Use `offer/poll/peek` (return null/false) instead of `add/remove/element` (throw exceptions) unless you want the exception.

Level 2

Core Techniques

The patterns that turn slow $O(n^2)$ answers into fast ones. Most Easy and Medium questions live here.

In this level: Two Pointers · Sliding Window · Prefix Sum & Difference Array · Sorting Algorithms · Binary Search · Monotonic Stack & Queue · Intervals & Sweep Line

Two Pointers

Level 2 · Core Techniques · ≈ 2 sessions

In simple words

*Use **two indexes** that move through the data in a smart way, so you avoid a nested loop. This usually turns **$O(n^2)$ into $O(n)$** with $O(1)$ extra space.*

*Three common shapes: **opposite ends** (left and right move toward each other), **same direction** (a slow writer and a fast reader), and **two arrays** (one pointer in each, like merging).*

> **Think of it like...**
>
> *Two people searching a bookshelf from both ends and walking toward each other — they cover it in one pass.*

Key ideas

- *Works best when the array is **sorted** (or you can sort it) — sorting gives you a rule for which pointer to move.*
- *For pair sum: sum too small → move left forward; sum too big → move right backward.*
- ***3Sum** = sort + fix one number + two pointers on the rest. Skip duplicates to avoid repeated triplets.*
- *Trapping Rain Water: move the side with the smaller max — its water level is already decided.*

Cost sheet

Shape	Typical use
Opposite ends	Pair sum in sorted array, palindrome, container with most water
Slow / fast (same direction)	Remove duplicates in place, move zeroes, partition
One pointer per array	Merge two sorted arrays, intersection

Java code

```java
// Pair with target sum in a sorted array
int[] pairSum(int[] a, int target) {
    int l = 0, r = a.length - 1;
    while (l < r) {
        int sum = a[l] + a[r];
        if (sum == target) return new int[]{l, r};
        if (sum < target) l++; else r--;
    }
    return new int[]{-1, -1};
}
```

```java
// Remove duplicates in place (slow = write, fast = read)
int removeDuplicates(int[] a) {
    if (a.length == 0) return 0;
    int slow = 0;
    for (int fast = 1; fast < a.length; fast++)
        if (a[fast] != a[slow]) a[++slow] = a[fast];
    return slow + 1;
}
```

```java
// 3Sum: all unique triplets that sum to 0
List<List<Integer>> threeSum(int[] nums) {
    Arrays.sort(nums);
    List<List<Integer>> res = new ArrayList<>();
    for (int i = 0; i < nums.length - 2; i++) {
        if (i > 0 && nums[i] == nums[i - 1]) continue;      // skip duplicate
        int l = i + 1, r = nums.length - 1;
        while (l < r) {
            int sum = nums[i] + nums[l] + nums[r];
            if (sum < 0) l++;
            else if (sum > 0) r--;
            else {
                res.add(List.of(nums[i], nums[l], nums[r]));
                while (l < r && nums[l] == nums[l + 1]) l++;
                while (l < r && nums[r] == nums[r - 1]) r--;
                l++; r--;
            }
        }
    }
    return res;
}
```

Spot it when

- *Sorted array + 'find a pair / triplet'.*
- *'In place', 'O(1) extra space'.*
- *Palindrome checks, reversing.*

Practice list

- [ ] Valid Palindrome (E)
- [ ] Remove Duplicates from Sorted Array (E)
- [ ] Move Zeroes (E)
- [ ] Two Sum II - Input Array Is Sorted (M)
- [ ] Container With Most Water (M)
- [ ] 3Sum (M)
- [ ] Sort Colors (M)
- [ ] Trapping Rain Water (H)

★ **Interview tip:** If the array is sorted and the problem asks for pairs, mention two pointers (O(1) space) before hashing (O(n) space) and compare the trade-off.

Sliding Window

Level 2 · Core Techniques · ≈ 3 sessions

In simple words

A **window** is a continuous range `[left, right]`. Instead of recomputing every subarray from scratch, you **slide**: add the new item on the right, remove the old item on the left. That turns O(n·k) or O(n²) into **O(n)**.

Fixed window: size k never changes. **Variable window**: grow right; while the window breaks a rule, shrink left.

Think of it like...

Looking at a long train through a window of a moving car: one carriage enters the view as another leaves.

Key ideas

- Keywords: **contiguous** subarray / substring, **longest**, **shortest**, **at most K**, **exactly K**.
- Track what's inside the window with a count array, HashMap, or running sum.
- 'Exactly K' = atMost(K) − atMost(K − 1).
- **Warning**: if the array can contain **negative numbers**, a sum window doesn't work (shrinking may not reduce the sum). Use prefix sum + HashMap instead.

Cost sheet

Type	Loop shape
Fixed size k	add a[r]; if r ≥ k, remove a[r−k]; record answer
Longest valid window	expand r; while invalid → shrink l; record r−l+1
Shortest valid window	expand r; while valid → record, then shrink l

Java code

```java
// Fixed window: max sum of any k consecutive elements
int maxSumK(int[] a, int k) {
    int sum = 0, best = Integer.MIN_VALUE;
    for (int r = 0; r < a.length; r++) {
        sum += a[r];
        if (r >= k) sum -= a[r - k];      // drop the element that left
        if (r >= k - 1) best = Math.max(best, sum);
    }
    return best;
}
```

```java
// Variable window: longest substring without repeating characters
int lengthOfLongestSubstring(String s) {
    int[] count = new int[128];
    int l = 0, best = 0;
    for (int r = 0; r < s.length(); r++) {
        char c = s.charAt(r);
        count[c]++;
        while (count[c] > 1) count[s.charAt(l++)]--;   // shrink until valid
        best = Math.max(best, r - l + 1);
    }
    return best;
}
```

```java
// Shortest window: minimum length subarray with sum >= target (positives)
int minSubArrayLen(int target, int[] nums) {
    int l = 0, sum = 0, best = Integer.MAX_VALUE;
    for (int r = 0; r < nums.length; r++) {
        sum += nums[r];
        while (sum >= target) {
            best = Math.min(best, r - l + 1);
            sum -= nums[l++];
        }
    }
    return best == Integer.MAX_VALUE ? 0 : best;
}
```

Spot it when

- → *'Longest / shortest substring or subarray that...'*
- → *'Every window of size k'.*
- → *'At most K distinct characters'.*

Practice list

- ☐ *Maximum Average Subarray I* (E)
- ☐ *Longest Substring Without Repeating Characters* (M)
- ☐ *Longest Repeating Character Replacement* (M)
- ☐ *Permutation in String* (M)
- ☐ *Minimum Size Subarray Sum* (M)
- ☐ *Fruit Into Baskets* (M)
- ☐ *Minimum Window Substring* (H)
- ☐ *Sliding Window Maximum* (H)

★ **Interview tip:** Memorize one variable-window template and reuse it. Most window problems differ only in the "is the window valid?" check.

Prefix Sum & Difference Array

Level 2 · Core Techniques · ≈ 2 sessions

In simple words

Build `pre[i]` = sum of the first i elements once (O(n)). Then the sum of any range `[l, r]` is just `pre[r+1] − pre[l]` — **O(1) per query**.

A **difference array** is the reverse trick: apply many "add x to range [l, r]" updates in O(1) each, then rebuild the final array with one prefix pass.

Think of it like...

Your car's odometer. To know how far you drove between two towns, subtract the two readings — no need to re-drive the road.

Key ideas

- **Subarray sum equals K**: while scanning, if `prefix − K` was seen before, those earlier positions start valid subarrays. Store prefix counts in a HashMap, starting with `{0: 1}`.
- Works with negative numbers (unlike sliding window).
- Same idea works for XOR (prefix XOR), counts (prefix count of vowels), and products.
- 2D: `sum = P[r2+1][c2+1] − P[r1][c2+1] − P[r2+1][c1] + P[r1][c1]`.
- Remainder trick: 'subarray sum divisible by k' → store `prefix % k` in the map.

Cost sheet

Task	Cost
Build prefix array	O(n)
Range sum query	O(1)
Range add (difference array)	O(1) each + O(n) rebuild
2D prefix build / query	O(r·c) / O(1)

Java code

```java
// Build prefix sums (size n + 1 to avoid edge cases)
int[] buildPrefix(int[] a) {
    int[] pre = new int[a.length + 1];
    for (int i = 0; i < a.length; i++) pre[i + 1] = pre[i] + a[i];
    return pre;
}
int rangeSum(int[] pre, int l, int r) { return pre[r + 1] - pre[l]; }
```

```java
// Count subarrays with sum exactly k (handles negatives)
int subarraySum(int[] nums, int k) {
    Map<Integer, Integer> seen = new HashMap<>();
    seen.put(0, 1);
    int sum = 0, count = 0;
    for (int x : nums) {
        sum += x;
        count += seen.getOrDefault(sum - k, 0);
        seen.merge(sum, 1, Integer::sum);
    }
    return count;
}
```

```java
// Difference array: many range updates, then one rebuild
int[] applyUpdates(int n, int[][] updates) {  // each update = {l, r, value}
    int[] diff = new int[n + 1];
    for (int[] u : updates) {
        diff[u[0]] += u[2];
        diff[u[1] + 1] -= u[2];
    }
    int[] res = new int[n];
    int run = 0;
    for (int i = 0; i < n; i++) { run += diff[i]; res[i] = run; }
    return res;
}
```

Spot it when

- → *'Sum of a range', many queries, no updates.*
- → *'Number of subarrays with sum / XOR equal to K'.*
- → *'Add value to range [l, r] many times (bookings, car pooling).*

Practice list

- ☐ *Range Sum Query - Immutable* (E)
- ☐ *Find Pivot Index* (E)
- ☐ *Subarray Sum Equals K* (M)
- ☐ *Continuous Subarray Sum* (M)
- ☐ *Product of Array Except Self* (M)
- ☐ *Range Sum Query 2D - Immutable* (M)
- ☐ *Car Pooling* (M)
- ☐ *Corporate Flight Bookings* (M)

★ **Interview tip:** Use a prefix array of length n + 1 with `pre[0] = 0`. It removes the "what if l = 0" special case.

Sorting Algorithms

Level 2 · Core Techniques · ≈ 3 sessions

In simple words

Sorting puts items in order. You'll mostly call `Arrays.sort`, but interviewers still ask you to **explain or write merge sort and quick sort**, and many solutions start with "first, sort".

Stable sort = equal items keep their original order (matters when sorting objects by one field).

Think of it like...

Merge sort is like sorting exam papers by splitting the pile in half, having two helpers sort each half, then merging the two sorted piles.

Key ideas

- **Insertion sort** is great for small or nearly sorted data (TimSort uses it inside).
- **Merge sort**: divide in half, sort both, merge. Always $O(n \log n)$. Used for linked lists and 'count inversions'.
- **Quick sort**: pick a pivot, put smaller on the left and bigger on the right, recurse. Random pivot avoids the $O(n^2)$ worst case.
- **Quickselect** finds the k-th smallest in $O(n)$ average — no full sort needed.
- **Counting / bucket sort**: when values are in a small range (e.g. 0–100, letters).
- Comparison sorts can't beat $O(n \log n)$ in the worst case.

Cost sheet

Algorithm	Best	Average	Worst	Space	Stable
Bubble	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes
Quick	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	No
Heap	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	No
Counting	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Yes

Java code

```java
// Merge sort
void mergeSort(int[] a, int lo, int hi, int[] tmp) {
  if (lo >= hi) return;
  int mid = lo + (hi - lo) / 2;
  mergeSort(a, lo, mid, tmp);
  mergeSort(a, mid + 1, hi, tmp);
  int i = lo, j = mid + 1, k = lo;
  while (i <= mid && j <= hi) tmp[k++] = (a[i] <= a[j]) ? a[i++] : a[j++];
  while (i <= mid) tmp[k++] = a[i++];
  while (j <= hi)  tmp[k++] = a[j++];
  for (k = lo; k <= hi; k++) a[k] = tmp[k];
}
```

```java
// Quick sort (Lomuto partition, random pivot)
Random rnd = new Random();
void quickSort(int[] a, int lo, int hi) {
  if (lo >= hi) return;
  int p = partition(a, lo, hi);
  quickSort(a, lo, p - 1);
  quickSort(a, p + 1, hi);
}
int partition(int[] a, int lo, int hi) {
  swap(a, lo + rnd.nextInt(hi - lo + 1), hi);  // random pivot to the end
  int pivot = a[hi], i = lo;
  for (int j = lo; j < hi; j++)
    if (a[j] < pivot) swap(a, i++, j);
  swap(a, i, hi);
  return i;
}
void swap(int[] a, int i, int j) { int t = a[i]; a[i] = a[j]; a[j] = t; }
```

```java
// Custom sorting: people by age, then by name
people.sort(Comparator.comparingInt(Person::age).thenComparing(Person::name));
```

Spot it when

- 'Merge', 'overlapping', 'closest', 'k-th largest' — sorting often simplifies the rest.
- 'Arrange to form the largest number' → custom comparator.

Practice list

- ☐ Merge Sorted Array (E)
- ☐ Sort an Array (M)
- ☐ Sort Colors (M)
- ☐ Kth Largest Element in an Array (M)
- ☐ Merge Intervals (M)
- ☐ Largest Number (M)
- ☐ Count of Smaller Numbers After Self (H)
- ☐ Reverse Pairs (H)

★ **Interview tip:** Be able to write merge sort and quick sort from memory in under 10 minutes — they are common warm-ups and appear in Java trivia (TimSort vs dual-pivot quicksort).

Binary Search

Level 2 · Core Techniques · ≈ 3 sessions

In simple words

In a **sorted** (or "monotonic") search space, look at the middle. The answer is either in the left half or the right half — **throw the other half away**. Each step halves the work: **O(log n)**.

The big idea for interviews: binary search works on anything shaped like `FFFFTTT`. You're looking for the **first T**. That covers sorted arrays, rotated arrays, and even "the smallest answer that works".

Think of it like...

Guessing a number between 1 and 100 when someone says "higher" or "lower". You never need more than 7 guesses.

Key ideas

- Compute mid as `lo + (hi - lo) / 2` to avoid integer overflow.
- Pick one template and always use it. The 'first true' template below never loops forever.
- **Binary search on the answer**: when the question says *minimize the maximum* or *maximize the minimum* (ship capacity, eating speed, split array), search over possible answers and write a greedy `canDo(x)` check.
- Complexity for search-on-answer: O(log(range) × cost of check).

Cost sheet

Variant	What you search
Classic	Exact target in sorted array
Lower bound	First index with a[i] ≥ target
Rotated array	Decide which half is sorted, then check if target is inside it
Binary search on answer	Smallest value x where canDo(x) is true
2D matrix	Treat rows × cols as one sorted list

Java code

```java
// Template: first index where condition becomes true (lower bound)
int lowerBound(int[] a, int target) {
    int lo = 0, hi = a.length;         // answer is in [lo, hi]
    while (lo < hi) {
        int mid = lo + (hi - lo) / 2;
        if (a[mid] >= target) hi = mid;  // mid could be the answer
        else lo = mid + 1;
    }
    return lo;                         // a.length if not found
}
```

```java
// Search in rotated sorted array
int search(int[] a, int target) {
    int lo = 0, hi = a.length - 1;
    while (lo <= hi) {
        int mid = lo + (hi - lo) / 2;
        if (a[mid] == target) return mid;
        if (a[lo] <= a[mid]) {                  // left half sorted
            if (a[lo] <= target && target < a[mid]) hi = mid - 1;
            else lo = mid + 1;
        } else {                                // right half sorted
            if (a[mid] < target && target <= a[hi]) lo = mid + 1;
            else hi = mid - 1;
        }
    }
    return -1;
}
```

```java
// Binary search on the answer: Koko eating bananas
int minEatingSpeed(int[] piles, int h) {
    int lo = 1, hi = Arrays.stream(piles).max().getAsInt();
    while (lo < hi) {
        int speed = lo + (hi - lo) / 2;
        if (hoursNeeded(piles, speed) <= h) hi = speed;  // fast enough
        else lo = speed + 1;
    }
    return lo;
}
long hoursNeeded(int[] piles, int speed) {
    long hours = 0;
    for (int p : piles) hours += (p + speed - 1) / speed;  // ceiling division
    return hours;
}
```

Spot it when

- Sorted input and O(log n) expected.
- 'Minimum capacity / speed / days such that...'
- Answer range is huge (up to 10^9) but checking one guess is easy.

Practice list

- [ ] Binary Search (E)
- [ ] Search Insert Position (E)
- [ ] First Bad Version (E)
- [ ] Find First and Last Position of Element in Sorted Array (M)
- [ ] Search in Rotated Sorted Array (M)
- [ ] Find Minimum in Rotated Sorted Array (M)
- [ ] Search a 2D Matrix (M)
- [ ] Koko Eating Bananas (M)
- [ ] Capacity To Ship Packages Within D Days (M)
- [ ] Split Array Largest Sum (H)
- [ ] Median of Two Sorted Arrays (H)

★ **Interview tip:** "Minimize the maximum" or "maximize the minimum" is a strong hint for binary search on the answer. Say it — it's a recognised senior-level insight.

Monotonic Stack & Queue

Level 2 · Core Techniques · ≈ 2 sessions

In simple words

A **monotonic stack** keeps its items always increasing (or always decreasing). When a new item breaks the order, you pop items — and at that moment you've found the **"next greater / next smaller" element** for each popped item.

A **monotonic deque** does the same for a sliding window, giving the window's max or min in O(1).

Think of it like...

People standing in a line, all looking right. A tall person arriving blocks the view of everyone shorter in front of them — those shorter people have found their "next taller person".

Key ideas

- Store **indexes** in the stack, not values — you usually need distances or widths.
- Each element is pushed once and popped once → **O(n)** total, even with a while-loop inside the for-loop.
- Largest rectangle in histogram: for each bar, find the previous and next smaller bars — width between them × height.
- Circular arrays: loop twice (`i % n`).

Cost sheet

Goal	Stack order	Scan
Next greater element	Decreasing	left → right
Next smaller element	Increasing	left → right
Previous greater	Decreasing	left → right (answer = top before push)
Sliding window max	Decreasing deque	pop front when out of window

Java code

```java
// Daily Temperatures: days until a warmer day
int[] dailyTemperatures(int[] t) {
    int[] ans = new int[t.length];
    Deque<Integer> st = new ArrayDeque<>();        // indexes, temps decreasing
    for (int i = 0; i < t.length; i++) {
        while (!st.isEmpty() && t[i] > t[st.peek()]) {
            int j = st.pop();
            ans[j] = i - j;                    // i is j's next warmer day
        }
        st.push(i);
    }
    return ans;
}
```

```java
// Sliding window maximum with a monotonic deque
int[] maxSlidingWindow(int[] a, int k) {
    int[] res = new int[a.length - k + 1];
    Deque<Integer> dq = new ArrayDeque<>();        // indexes, values decreasing
    for (int i = 0; i < a.length; i++) {
        if (!dq.isEmpty() && dq.peekFirst() <= i - k) dq.pollFirst();   // out of window
        while (!dq.isEmpty() && a[dq.peekLast()] < a[i]) dq.pollLast();  // useless now
        dq.offerLast(i);
        if (i >= k - 1) res[i - k + 1] = a[dq.peekFirst()];
    }
    return res;
}
```

```java
// Largest rectangle in histogram
int largestRectangleArea(int[] h) {
    Deque<Integer> st = new ArrayDeque<>();
    int best = 0;
    for (int i = 0; i <= h.length; i++) {
        int cur = (i == h.length) ? 0 : h[i];      // sentinel flushes the stack
        while (!st.isEmpty() && cur < h[st.peek()]) {
            int height = h[st.pop()];
            int left = st.isEmpty() ? -1 : st.peek();
            best = Math.max(best, height * (i - left - 1));
        }
        st.push(i);
    }
    return best;
}
```

Spot it when

- → *'Next greater / smaller', 'how many days until', 'span'.*
- → *'Maximum in every window of size k'.*
- → *Histogram / rectangle / 'remove k digits to make smallest'.*

Practice list

- [ ] Next Greater Element I (E)
- [ ] Daily Temperatures (M)
- [ ] Online Stock Span (M)
- [ ] Next Greater Element II (M)
- [ ] Sum of Subarray Minimums (M)
- [ ] Remove K Digits (M)
- [ ] Largest Rectangle in Histogram (H)
- [ ] Sliding Window Maximum (H)
- [ ] Maximal Rectangle (H)

★ **Interview tip:** Explain the amortized O(n): "every index enters and leaves the stack at most once." Interviewers often probe this.

Intervals & Sweep Line

Level 2 · Core Techniques · ≈ 2 sessions

In simple words

Interval problems give you pairs `[start, end]` — meetings, bookings, ranges. The first move is almost always ***sort by start*** (or by end for greedy selection).

Sweep line: turn each interval into two events (+1 at start, −1 at end), sort the events, and walk through time keeping a running count.

Think of it like...

A meeting-room calendar. Sort meetings by start time and you can see overlaps as you read top to bottom.

Key ideas

- Two intervals overlap when `a.start <= b.end && b.start <= a.end`.
- Ask whether touching intervals like [1,2] and [2,3] count as overlapping.
- Min-heap of end times: the heap size is the number of rooms in use.
- `TreeMap<Integer,Integer>` as a sweep line handles 'My Calendar' style online bookings.

Cost sheet

Question	Approach
Merge overlapping	Sort by start; extend the last merged interval
Min meeting rooms	Sort starts & ends, or a min-heap of end times
Max non-overlapping	Greedy: sort by end, keep earliest finishing
Insert an interval	Add all before, merge overlapping, add all after

Java code

```java
// Merge intervals
int[][] merge(int[][] intervals) {
    Arrays.sort(intervals, (a, b) -> Integer.compare(a[0], b[0]));
    List<int[]> res = new ArrayList<>();
    for (int[] cur : intervals) {
        if (res.isEmpty() || res.get(res.size() - 1)[1] < cur[0]) {
            res.add(cur);                          // no overlap
        } else {
            int[] last = res.get(res.size() - 1);
            last[1] = Math.max(last[1], cur[1]);           // merge
        }
    }
    return res.toArray(new int[0][]);
}
```

```java
// Meeting Rooms II: minimum rooms needed
int minMeetingRooms(int[][] meetings) {
    Arrays.sort(meetings, (a, b) -> Integer.compare(a[0], b[0]));
    PriorityQueue<Integer> ends = new PriorityQueue<>();      // end times in use
    for (int[] m : meetings) {
        if (!ends.isEmpty() && ends.peek() <= m[0]) ends.poll(); // reuse a room
        ends.offer(m[1]);
    }
    return ends.size();
}
```

Spot it when

- → *Input is a list of [start, end].*
- → *'Overlapping', 'conflicts', 'free time', 'rooms', 'arrows'.*

Practice list

- ☐ *Meeting Rooms* (E)
- ☐ *Merge Intervals* (M)
- ☐ *Insert Interval* (M)
- ☐ *Non-overlapping Intervals* (M)
- ☐ *Meeting Rooms II* (M)
- ☐ *Minimum Number of Arrows to Burst Balloons* (M)
- ☐ *Interval List Intersections* (M)
- ☐ *Employee Free Time* (H)

★ ***Interview tip:*** *Ask whether the end is inclusive or exclusive before coding — it changes* `<` *to* `<=` *in exactly the place bugs hide.*

Level 3

Trees & Heaps

Hierarchical data, sorted trees, priority queues and prefix trees.

In this level: Binary Trees & Traversals · Binary Search Trees (BST) · Heap / Priority Queue · Trie (Prefix Tree)

Binary Trees & Traversals

Level 3 · Trees & Heaps · ≈ 4 sessions

In simple words

A **tree** is a hierarchy: one **root** at the top, each node has children, nodes with no children are **leaves**. In a **binary tree** every node has at most two children: left and right.

Height = longest path from a node down to a leaf. **Depth** = distance from the root.

Almost every tree problem is solved by asking: *"What does this node need from its left and right children?"* — then letting recursion do the rest.

> **Think of it like...**
>
> A company org chart: the CEO at the top, managers below, and individual contributors as leaves.

Key ideas

- Every traversal visits each node once: **O(n) time**, **O(h) space** for recursion (h = height; n in the worst, skewed case).
- **Two recursion styles**: return a value up (height, sum) or pass information down (current path, bounds).
- **Global answer pattern**: diameter and max path sum — return one thing to the parent but update a global best with another.
- Tree kinds: **full** (0 or 2 children), **complete** (filled left to right — heaps), **perfect**, **balanced** (heights differ by ≤ 1).
- **LCA** (lowest common ancestor): if p and q are found in different subtrees, the current node is the answer.
- Build a tree from preorder + inorder: preorder gives the root; inorder splits left and right.

Cost sheet

Traversal	Order	Typical use
Preorder (DFS)	Root → Left → Right	Copy / serialize a tree
Inorder (DFS)	Left → Root → Right	Sorted order in a BST
Postorder (DFS)	Left → Right → Root	Heights, deleting, 'answer from children'
Level order (BFS)	Level by level with a queue	Right-side view, min depth, zigzag

Java code

```
class TreeNode {
  int val; TreeNode left, right;
  TreeNode(int v) { val = v; }
}
```

```
// DFS traversals
void inorder(TreeNode n, List<Integer> out) {
  if (n == null) return;
  inorder(n.left, out);
  out.add(n.val);
  inorder(n.right, out);
}
```

```
// BFS: level order
List<List<Integer>> levelOrder(TreeNode root) {
  List<List<Integer>> res = new ArrayList<>();
  if (root == null) return res;
  Queue<TreeNode> q = new ArrayDeque<>();
  q.offer(root);
  while (!q.isEmpty()) {
    int size = q.size();              // nodes in this level
    List<Integer> level = new ArrayList<>();
    for (int i = 0; i < size; i++) {
      TreeNode n = q.poll();
      level.add(n.val);
      if (n.left != null)  q.offer(n.left);
      if (n.right != null) q.offer(n.right);
    }
    res.add(level);
  }
  return res;
}
```

```
// Max depth
int maxDepth(TreeNode n) {
  return n == null ? 0 : 1 + Math.max(maxDepth(n.left), maxDepth(n.right));
}
```

```java
// Diameter: return height, update global best
int best = 0;
int diameterOfBinaryTree(TreeNode root) { height(root); return best; }
int height(TreeNode n) {
    if (n == null) return 0;
    int l = height(n.left), r = height(n.right);
    best = Math.max(best, l + r);          // path through n
    return 1 + Math.max(l, r);
}
```

```java
// Lowest common ancestor
TreeNode lca(TreeNode n, TreeNode p, TreeNode q) {
    if (n == null || n == p || n == q) return n;
    TreeNode l = lca(n.left, p, q), r = lca(n.right, p, q);
    if (l != null && r != null) return n;      // split here
    return l != null ? l : r;
}
```

Spot it when

- → *Input is a* `TreeNode`.
- → *'Level', 'depth', 'path', 'ancestor', 'subtree'.*

Practice list

- ☐ *Maximum Depth of Binary Tree* (E)
- ☐ *Invert Binary Tree* (E)
- ☐ *Same Tree* (E)
- ☐ *Diameter of Binary Tree* (E)
- ☐ *Balanced Binary Tree* (E)
- ☐ *Binary Tree Level Order Traversal* (M)
- ☐ *Binary Tree Right Side View* (M)
- ☐ *Lowest Common Ancestor of a Binary Tree* (M)
- ☐ *Construct Binary Tree from Preorder and Inorder Traversal* (M)
- ☐ *Path Sum II* (M)
- ☐ *Binary Tree Maximum Path Sum* (H)
- ☐ *Serialize and Deserialize Binary Tree* (H)

★ ***Interview tip:*** *State the space cost honestly: O(h), which is O(log n) for a balanced tree but O(n) for a skewed one.*

Binary Search Trees (BST)

Level 3 · Trees & Heaps · ≈ 2 sessions

In simple words

A BST is a binary tree with one rule: for every node, ***everything on the left is smaller*** *and* ***everything on the right is bigger****. That rule lets you search like binary search — go left or right at each step.*

An ***inorder traversal*** *of a BST gives values in* ***sorted order****.*

Think of it like...

A "guess the number" game built into the tree: at each node you're told "smaller, go left" or "bigger, go right".

Key ideas

- Self-balancing BSTs (AVL, **Red-Black**) keep height at O(log n). Java's `TreeMap` / `TreeSet` are red-black trees.
- **Validate BST**: pass allowed (min, max) bounds down — comparing a node only with its children is the classic wrong answer.
- **Delete**: 0 children → remove; 1 child → replace with the child; 2 children → replace value with the inorder successor, then delete that successor.
- k-th smallest = the k-th node in inorder traversal.
- LCA in a BST is easier: walk down until p and q fall on different sides.

Cost sheet

Operation	Balanced	Skewed (worst)
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)
Min / max	O(log n)	O(n)
Inorder (sorted)	O(n)	O(n)

Java code

```
TreeNode searchBST(TreeNode n, int target) {
  while (n != null && n.val != target)
    n = target < n.val ? n.left : n.right;
  return n;
}
```

```
TreeNode insert(TreeNode n, int v) {
  if (n == null) return new TreeNode(v);
  if (v < n.val) n.left = insert(n.left, v);
  else           n.right = insert(n.right, v);
  return n;
}
```

```
// Validate with bounds (long to handle Integer.MIN/MAX values)
boolean isValidBST(TreeNode root) { return valid(root, Long.MIN_VALUE, Long.MAX_VALUE); }
boolean valid(TreeNode n, long min, long max) {
  if (n == null) return true;
  if (n.val <= min || n.val >= max) return false;
  return valid(n.left, min, n.val) && valid(n.right, n.val, max);
}
```

```java
// k-th smallest with iterative inorder
int kthSmallest(TreeNode root, int k) {
    Deque<TreeNode> st = new ArrayDeque<>();
    TreeNode cur = root;
    while (true) {
        while (cur != null) { st.push(cur); cur = cur.left; }
        cur = st.pop();
        if (--k == 0) return cur.val;
        cur = cur.right;
    }
}
```

Spot it when

- → *The tree is a BST → think 'sorted and 'go left or right'.*
- → *'k-th smallest', 'closest value', 'range sum in BST'.*

Practice list

- ☐ *Search in a Binary Search Tree* (E)
- ☐ *Convert Sorted Array to Binary Search Tree* (E)
- ☐ *Validate Binary Search Tree* (M)
- ☐ *Kth Smallest Element in a BST* (M)
- ☐ *Lowest Common Ancestor of a Binary Search Tree* (M)
- ☐ *Delete Node in a BST* (M)
- ☐ *Binary Search Tree Iterator* (M)
- ☐ *Recover Binary Search Tree* (M)

★ **Interview tip:** *When asked "how would you keep it fast with many inserts?", mention self-balancing trees and that Java's TreeMap is a red-black tree.*

Heap / Priority Queue

Level 3 · Trees & Heaps · ≈ 3 sessions

In simple words

*A **heap** gives you the **smallest (min-heap) or largest (max-heap) item instantly**, and lets you add or remove items in O(log n). It is a complete binary tree stored inside an array.*

Java: `PriorityQueue` *is a min-heap by default.*

Think of it like...

A hospital emergency room: the most urgent patient is always seen next, no matter who arrived first.

Key ideas

- ✱ *Array layout: children of index i are* `2i+1` *and* `2i+2`*; parent is* `(i-1)/2`*.*

- **Top-K largest**: keep a **min-heap of size K**; if it grows beyond K, remove the smallest. Result: O(n log K).
- **Two heaps** (max-heap for lower half, min-heap for upper half) → running median.
- **K-way merge**: put the head of each list in a heap; always take the smallest.
- Scheduling: 'always pick the task with the highest priority / earliest end.
- Dijkstra's shortest path uses a min-heap (Level 5).

Cost sheet

Operation	Time
peek (min or max)	O(1)
offer (insert)	O(log n)
poll (remove top)	O(log n)
Build heap from n items	O(n)
remove(arbitrary) / contains	O(n)

Java code

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

```
// Top K frequent elements
int[] topKFrequent(int[] nums, int k) {
    Map<Integer, Integer> freq = new HashMap<>();
    for (int x : nums) freq.merge(x, 1, Integer::sum);
    PriorityQueue<Map.Entry<Integer, Integer>> pq =
        new PriorityQueue<>((a, b) -> Integer.compare(a.getValue(), b.getValue()));
    for (Map.Entry<Integer, Integer> e : freq.entrySet()) {
        pq.offer(e);
        if (pq.size() > k) pq.poll();
    }
    int[] res = new int[k];
    for (int i = 0; i < k; i++) res[i] = pq.poll().getKey();
    return res;
}
```

```java
// Running median with two heaps
class MedianFinder {
  PriorityQueue<Integer> low  = new PriorityQueue<>(Collections.reverseOrder()); // max-heap
  PriorityQueue<Integer> high = new PriorityQueue<>();                           // min-heap
  void addNum(int x) {
    low.offer(x);
    high.offer(low.poll());                // keep order between halves
    if (high.size() > low.size()) low.offer(high.poll());  // balance sizes
  }
  double findMedian() {
    return low.size() > high.size() ? low.peek() : (low.peek() + high.peek()) / 2.0;
  }
}
```

Spot it when

- → *'K largest / smallest / closest / most frequent'.*
- → *'Median of a stream'.*
- → *'Merge K sorted...'*
- → *'Always process the cheapest / earliest next'.*

Practice list

- ☐ *Kth Largest Element in a Stream* (E)
- ☐ *Last Stone Weight* (E)
- ☐ *Kth Largest Element in an Array* (M)
- ☐ *Top K Frequent Elements* (M)
- ☐ *K Closest Points to Origin* (M)
- ☐ *Task Scheduler* (M)
- ☐ *Find Median from Data Stream* (H)
- ☐ *Merge k Sorted Lists* (H)
- ☐ *IPO* (H)

★ **Interview tip:** *For "top K largest", a min-heap of size K sounds backwards — explain why (the smallest of the best K sits on top, ready to be kicked out).*

Trie (Prefix Tree)

Level 3 · Trees & Heaps · ≈ 2 sessions

In simple words

*A trie stores words **character by character** along paths from the root. Words that share a prefix share the same path, so checking "does any word start with pre?" takes only as long as the prefix.*

Think of it like...

Your phone keyboard's autocomplete: type "ca" and it instantly knows about "cat", "car", and "cake".

Key ideas

- Each node has `children[26]` (or a HashMap for large alphabets) and an `isEnd` flag.
- Wildcard search (`.` matches anything): DFS into all children at that position.
- **Word Search II**: build a trie of all words, then DFS the grid once — much faster than searching each word separately.
- **Bitwise trie** (children[2]) solves 'maximum XOR of two numbers'.

Cost sheet

Operation	Time	Note
insert(word)	O(L)	L = word length
search(word)	O(L)	must end at a word marker
startsWith(prefix)	O(L)	path just needs to exist
Space	O(total characters × alphabet)	can be heavy

Java code

```
class Trie {
  private static class Node {
    Node[] next = new Node[26];
    boolean isEnd;
  }
  private final Node root = new Node();
```

```
  void insert(String word) {
    Node cur = root;
    for (char c : word.toCharArray()) {
      int i = c - 'a';
      if (cur.next[i] == null) cur.next[i] = new Node();
      cur = cur.next[i];
    }
    cur.isEnd = true;
  }
  boolean search(String word)      { Node n = find(word);  return n != null && n.isEnd; }
  boolean startsWith(String prefix) { return find(prefix) != null; }
```

```
  private Node find(String s) {
    Node cur = root;
    for (char c : s.toCharArray()) {
      cur = cur.next[c - 'a'];
      if (cur == null) return null;
    }
    return cur;
  }
}
```

Spot it when

- → 'Prefix', 'autocomplete', 'dictionary of words'.
- → Searching many words at once in a grid or text.

Practice list

- ☐ Implement Trie (Prefix Tree) (M)
- ☐ Design Add and Search Words Data Structure (M)
- ☐ Replace Words (M)
- ☐ Search Suggestions System (M)
- ☐ Maximum XOR of Two Numbers in an Array (M)
- ☐ Word Search II (H)

★ **Interview tip:** Mention the memory trade-off: a trie is fast but can use a lot of memory; a HashMap of children saves space for sparse alphabets.

Level 4

Backtracking & Greedy

Explore every option smartly — or prove you only need the best-looking one.

In this level: Backtracking · Greedy Algorithms

Backtracking

Level 4 · Backtracking & Greedy · ≈ 4 sessions

In simple words

Backtracking builds an answer **one choice at a time**. If a choice leads to a dead end, you **undo it** and try the next choice. The rhythm is always:

Choose → Explore → Un-choose.

It explores a "decision tree" of all possibilities, so it is exponential — but **pruning** (stopping early when a path can't work) makes it practical.

> **Think of it like...**
>
> Walking through a maze: at a dead end you walk back to the last junction and try the next path.

Key ideas

- Always add a **copy** of the path to results: `res.add(new ArrayList<>(path))`. Adding `path` itself is a classic bug.
- **Subsets / combinations**: pass a `start` index so you don't reuse earlier items.
- **Permutations**: use a `used[]` array because order matters.
- **Duplicates in input**: sort first, then skip `if (i > start && nums[i] == nums[i-1])`.
- **Grid search**: mark a cell visited before recursing, unmark after.
- Prune early: if the running sum exceeds the target (with sorted input) → `break`.

Cost sheet

Problem	Typical time
Subsets	$O(n \cdot 2^n)$
Permutations	$O(n \cdot n!)$
Combination sum	Exponential, depends on target
N-Queens	$O(n!)$
Word search in grid	$O(cells \cdot 3^L)$

Java code

```java
// Universal template
void backtrack(State state, List<Result> res) {
    if (isComplete(state)) { res.add(copyOf(state)); return; }
    for (Choice c : choices(state)) {
        if (!isValid(state, c)) continue;   // prune
        apply(state, c);                    // choose
        backtrack(state, res);              // explore
        undo(state, c);                     // un-choose
    }
}
```

```java
// Subsets
List<List<Integer>> subsets(int[] nums) {
    List<List<Integer>> res = new ArrayList<>();
    dfs(nums, 0, new ArrayList<>(), res);
    return res;
}
void dfs(int[] nums, int start, List<Integer> path, List<List<Integer>> res) {
    res.add(new ArrayList<>(path));
    for (int i = start; i < nums.length; i++) {
        path.add(nums[i]);
        dfs(nums, i + 1, path, res);
        path.remove(path.size() - 1);
    }
}
```

```java
// Permutations
void permute(int[] nums, boolean[] used, List<Integer> path, List<List<Integer>> res) {
    if (path.size() == nums.length) { res.add(new ArrayList<>(path)); return; }
    for (int i = 0; i < nums.length; i++) {
        if (used[i]) continue;
        used[i] = true;  path.add(nums[i]);
        permute(nums, used, path, res);
        used[i] = false; path.remove(path.size() - 1);
    }
}
```

```java
// Combination Sum (numbers can be reused), with pruning
void combo(int[] c, int start, int remain, List<Integer> path, List<List<Integer>> res) {
    if (remain == 0) { res.add(new ArrayList<>(path)); return; }
    for (int i = start; i < c.length; i++) {
        if (c[i] > remain) break;        // c is sorted -> stop early
        path.add(c[i]);
        combo(c, i, remain - c[i], path, res);   // i, not i + 1: reuse allowed
        path.remove(path.size() - 1);
    }
}
```

Spot it when

- → *'Return **all** combinations / permutations / subsets / partitions'.*
- → *Small n (≤ 15–20).*
- → *Puzzles: N-Queens, Sudoku, word search.*

Practice list

- ☐ Subsets (M)
- ☐ Subsets II (M)
- ☐ Permutations (M)
- ☐ Combination Sum (M)
- ☐ Combination Sum II (M)
- ☐ Letter Combinations of a Phone Number (M)
- ☐ Generate Parentheses (M)
- ☐ Palindrome Partitioning (M)
- ☐ Word Search (M)
- ☐ N-Queens (H)
- ☐ Sudoku Solver (H)

★ **Interview tip:** *If the question asks for the **count** or the **best** answer (not the full list), backtracking is probably too slow — look for DP.*

Greedy Algorithms

Level 4 · Backtracking & Greedy · ≈ 3 sessions

In simple words

*A greedy algorithm makes the **best-looking choice at each step** and never goes back. It is simple and fast — but it is only correct when **local best choices lead to the global best answer**.*

Before trusting greedy, try to break it with a small counter-example. If you can't, explain why it works (usually "swapping to my choice never makes the answer worse").

Think of it like...

Paying cash with the fewest notes: take the biggest note that fits, again and again. It works for Indian rupee notes — but not for every coin system.

Key ideas

- Greedy **fails** for coins {1, 3, 4} and amount 6: greedy picks 4+1+1 (3 coins), but 3+3 (2 coins) is better → needs DP.
- 0/1 knapsack (can't split items) also needs DP; fractional knapsack is greedy.
- Greedy often starts with **sorting** by the right key (end time, ratio, deadline).
- Greedy + heap is a common pairing (task scheduling, IPO, meeting rooms).
- Complexity is usually O(n) or O(n log n) because of the sort.

Cost sheet

Problem	Greedy rule
Activity selection / non-overlapping intervals	Pick the one that ends earliest
Jump game	Track the farthest index reachable
Gas station	If the tank goes negative, restart from the next station
Partition labels	Extend the cut to each letter's last position
Fractional knapsack	Highest value per kg first

Java code

```java
// Jump Game: can we reach the last index?
boolean canJump(int[] nums) {
    int farthest = 0;
    for (int i = 0; i < nums.length; i++) {
        if (i > farthest) return false;        // stuck before i
        farthest = Math.max(farthest, i + nums[i]);
    }
    return true;
}
```

```java
// Jump Game II: minimum jumps (BFS by ranges)
int jump(int[] nums) {
    int jumps = 0, curEnd = 0, farthest = 0;
    for (int i = 0; i < nums.length - 1; i++) {
        farthest = Math.max(farthest, i + nums[i]);
        if (i == curEnd) { jumps++; curEnd = farthest; }
    }
    return jumps;
}
```

```java
// Non-overlapping intervals: minimum removals
int eraseOverlapIntervals(int[][] iv) {
    Arrays.sort(iv, (a, b) -> Integer.compare(a[1], b[1]));  // by end
    int kept = 0, end = Integer.MIN_VALUE;
    for (int[] x : iv) {
        if (x[0] >= end) { kept++; end = x[1]; }
    }
    return iv.length - kept;
}
```

```java
// Gas Station
int canCompleteCircuit(int[] gas, int[] cost) {
    int total = 0, tank = 0, start = 0;
    for (int i = 0; i < gas.length; i++) {
        total += gas[i] - cost[i];
        tank  += gas[i] - cost[i];
        if (tank < 0) { start = i + 1; tank = 0; }
    }
    return total >= 0 ? start : -1;
}
```

Spot it when

- → *'Minimum number of...', 'maximum number of non-overlapping...'.*
- → *A sort by one key seems to make the choice obvious.*
- → *Constraints are large (10^5+) so DP tables won't fit.*

Practice list

- ☐ *Assign Cookies* (E)
- ☐ *Lemonade Change* (E)
- ☐ *Jump Game* (M)
- ☐ *Jump Game II* (M)
- ☐ *Gas Station* (M)
- ☐ *Partition Labels* (M)
- ☐ *Hand of Straights* (M)
- ☐ *Task Scheduler* (M)
- ☐ *Candy* (H)

★ **Interview tip:** *Say: "I think greedy works because... let me test a small counter-example."*
Interviewers reward the check more than a lucky guess.

Level 5

Graphs

Networks of connections: traversal, ordering, shortest paths and grouping.

In this level: Graphs: BFS & DFS · Topological Sort · Shortest Path Algorithms · Union-Find & Minimum Spanning Tree

Graphs: BFS & DFS

Level 5 · Graphs · ≈ 5 sessions

In simple words

A graph is a set of **nodes (vertices)** connected by **edges**. Edges can be **directed** (one-way) or **undirected**, **weighted** or not. A tree is just a graph with no cycles.

BFS (breadth-first) explores in rings using a queue — it finds the **shortest path when all edges cost the same**.
DFS (depth-first) goes as deep as possible first using recursion or a stack — great for connectivity, cycles, and paths.

Think of it like...

Cities joined by roads. BFS is a ripple spreading from a dropped stone; DFS is a hiker who follows one trail to its end before returning.

Key ideas

- Use an **adjacency list**: `List<List<Integer>>`. Use a matrix only for dense graphs.
- A **grid** is a hidden graph: each cell connects to up/down/left/right. Use a direction array.
- Always track **visited**, and mark it **when you add to the queue** (not when you pop) to avoid duplicates.
- **Connected components**: start a DFS/BFS from every unvisited node; count the starts.
- **Cycle detection**: undirected → a visited neighbour that isn't your parent; directed → a node currently on the recursion stack (3 colours: white/grey/black).
- **Bipartite**: try to 2-colour the graph with BFS; a conflict means not bipartite.
- **Multi-source BFS**: put all starting points in the queue at once (rotting oranges, distance to nearest 0).

Cost sheet

Item	Cost
BFS / DFS	O(V + E)
Adjacency list space	O(V + E)
Adjacency matrix space	O(V²)
Check edge (matrix)	O(1)
Grid BFS / DFS	O(rows × cols)

Java code

```
// Build an undirected adjacency list
List<List<Integer>> build(int n, int[][] edges) {
   List<List<Integer>> g = new ArrayList<>();
   for (int i = 0; i < n; i++) g.add(new ArrayList<>());
   for (int[] e : edges) { g.get(e[0]).add(e[1]); g.get(e[1]).add(e[0]); }
   return g;
}
```

```
// BFS: shortest number of edges from src to every node
int[] bfs(List<List<Integer>> g, int src) {
   int[] dist = new int[g.size()];
   Arrays.fill(dist, -1);
   Queue<Integer> q = new ArrayDeque<>();
   q.offer(src); dist[src] = 0;
   while (!q.isEmpty()) {
      int u = q.poll();
      for (int v : g.get(u)) {
         if (dist[v] == -1) {          // mark when enqueuing
            dist[v] = dist[u] + 1;
            q.offer(v);
         }
      }
   }
   return dist;
}
```

```java
// DFS on a grid: Number of Islands
int[][] DIRS = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};
int numIslands(char[][] grid) {
    int count = 0;
    for (int r = 0; r < grid.length; r++)
        for (int c = 0; c < grid[0].length; c++)
            if (grid[r][c] == '1') { count++; sink(grid, r, c); }
    return count;
}
void sink(char[][] g, int r, int c) {
    if (r < 0 || c < 0 || r >= g.length || c >= g[0].length || g[r][c] != '1') return;
    g[r][c] = '0';                    // mark visited
    for (int[] d : DIRS) sink(g, r + d[0], c + d[1]);
}
```

```java
// Directed cycle detection with colours: 0 = new, 1 = in progress, 2 = done
boolean hasCycle(int u, List<List<Integer>> g, int[] state) {
    state[u] = 1;
    for (int v : g.get(u)) {
        if (state[v] == 1) return true;      // back edge
        if (state[v] == 0 && hasCycle(v, g, state)) return true;
    }
    state[u] = 2;
    return false;
}
```

Spot it when

- → *'Network', 'connections', 'friends', 'islands', 'regions', 'grid'.*
- → *'Minimum number of steps / moves' with equal cost → BFS.*
- → *'Can you reach...', 'how many groups...' → DFS/BFS or Union-Find.*

Practice list

- ☐ *Flood Fill* (E)
- ☐ *Find if Path Exists in Graph* (E)
- ☐ *Number of Islands* (M)
- ☐ *Max Area of Island* (M)
- ☐ *Clone Graph* (M)
- ☐ *Rotting Oranges* (M)
- ☐ *Pacific Atlantic Water Flow* (M)
- ☐ *Surrounded Regions* (M)
- ☐ *Is Graph Bipartite?* (M)
- ☐ *Shortest Path in Binary Matrix* (M)
- ☐ *Number of Connected Components in an Undirected Graph* (M)
- ☐ *Word Ladder* (H)

★ ***Interview tip:*** *On large grids, recursive DFS can overflow the stack. Mention that you'd switch to BFS or an explicit stack in production.*

Topological Sort

Level 5 · Graphs · ≈ 2 sessions

In simple words

Topological sort orders tasks so that **every task comes after the tasks it depends on**. It only works on a **DAG** (directed graph with no cycles). If there's a cycle, no valid order exists.

Kahn's algorithm (BFS): repeatedly take tasks with no remaining prerequisites (in-degree 0).

> **Think of it like...**
>
> Getting dressed: socks before shoes, shirt before tie. Or a build tool deciding which modules to compile first.

Key ideas

- In-degree = number of arrows coming **into** a node.
- If the processed count is less than V, the graph has a **cycle** → e.g. 'courses can't be finished.
- Edge direction matters: for 'to take course a you need b', the edge is `b → a`.
- Alien Dictionary: compare adjacent words to find letter-order edges, then topo sort.
- Longest path in a DAG = topo order + DP.

Cost sheet

Method	Idea	Time
Kahn's (BFS)	Process in-degree 0 nodes; decrease neighbours	O(V + E)
DFS	Add node after all its children finish; reverse the list	O(V + E)

Java code

```
// Course Schedule II: return a valid order or empty array
int[] findOrder(int n, int[][] prereq) {
  List<List<Integer>> g = new ArrayList<>();
  for (int i = 0; i < n; i++) g.add(new ArrayList<>());
  int[] indeg = new int[n];
  for (int[] p : prereq) {       // p = {course, requiredCourse}
    g.get(p[1]).add(p[0]);
    indeg[p[0]]++;
  }
  Queue<Integer> q = new ArrayDeque<>();
  for (int i = 0; i < n; i++) if (indeg[i] == 0) q.offer(i);
```

```
    int[] order = new int[n];
    int idx = 0;
    while (!q.isEmpty()) {
        int u = q.poll();
        order[idx++] = u;
        for (int v : g.get(u))
            if (--indeg[v] == 0) q.offer(v);
    }
    return idx == n ? order : new int[0];   // cycle if not all processed
}
```

Spot it when

- → *'Prerequisites', 'dependencies', 'build order', 'must happen before'.*
- → *'Is it possible to finish all...' → cycle check.*

Practice list

- ☐ *Course Schedule* (M)
- ☐ *Course Schedule II* (M)
- ☐ *Find All Possible Recipes from Given Supplies* (M)
- ☐ *Parallel Courses* (M)
- ☐ *Sequence Reconstruction* (M)
- ☐ *Alien Dictionary* (H)
- ☐ *Longest Increasing Path in a Matrix* (H)

★ **Interview tip:** *Connect it to real work: microservice startup order, CI pipeline stages, or Maven module builds are all topological sorts.*

Shortest Path Algorithms

Level 5 · Graphs · ≈ 3 sessions

In simple words

*When edges have **different costs** (distance, time, price), plain BFS is not enough. Pick the algorithm by the kind of weights and whether you need one source or all pairs.*

***Dijkstra** is the one you must know by heart: always expand the closest unfinished node, using a min-heap.*

Think of it like...

Google Maps: from your location, it keeps extending the cheapest known route until it reaches your destination.

Key ideas

- ✱ *Dijkstra fails with **negative** edges — use Bellman-Ford.*
- ✱ *In Dijkstra, skip **stale** heap entries:* `if (d > dist[u]) continue;`

- Bellman-Ford: relax all edges V−1 times. If an edge can still be relaxed after that → negative cycle.
- 'Cheapest flight with at most K stops' → Bellman-Ford limited to K+1 rounds (copy the array each round).
- Grid with costs (min effort, swim in water) → Dijkstra on cells.

Cost sheet

Algorithm	Use when	Time
BFS	All edges cost the same	$O(V + E)$
0-1 BFS (deque)	Edge costs are only 0 or 1	$O(V + E)$
Dijkstra	Non-negative weights, one source	$O((V + E) \log V)$
Bellman-Ford	Negative weights; detect negative cycles; 'at most K edges'	$O(V \cdot E)$
Floyd-Warshall	All pairs, small V (≤ 400)	$O(V^3)$

Java code

```
// Dijkstra with an adjacency list of {neighbour, weight}
int[] dijkstra(List<List<int[]>> g, int src) {
    int n = g.size();
    int[] dist = new int[n];
    Arrays.fill(dist, Integer.MAX_VALUE);
    dist[src] = 0;
    PriorityQueue<int[]> pq = new PriorityQueue<>((a, b) -> Integer.compare(a[1], b[1]));
    pq.offer(new int[]{src, 0});
    while (!pq.isEmpty()) {
        int[] top = pq.poll();
        int u = top[0], d = top[1];
        if (d > dist[u]) continue;              // stale entry
        for (int[] e : g.get(u)) {
            int v = e[0], nd = d + e[1];
            if (nd < dist[v]) {
                dist[v] = nd;
                pq.offer(new int[]{v, nd});
            }
        }
    }
    return dist;
}
```

```java
// Bellman-Ford limited to k stops (Cheapest Flights Within K Stops)
int findCheapestPrice(int n, int[][] flights, int src, int dst, int k) {
    int[] dist = new int[n];
    Arrays.fill(dist, Integer.MAX_VALUE);
    dist[src] = 0;
    for (int round = 0; round <= k; round++) {
        int[] next = dist.clone();          // use last round's values only
        for (int[] f : flights) {
            if (dist[f[0]] == Integer.MAX_VALUE) continue;
            next[f[1]] = Math.min(next[f[1]], dist[f[0]] + f[2]);
        }
        dist = next;
    }
    return dist[dst] == Integer.MAX_VALUE ? -1 : dist[dst];
}
```

Spot it when

- → *'Minimum cost / time / effort' with weighted edges.*
- → *'Network delay', 'cheapest route', 'path with minimum...'.*

Practice list

- ☐ *Network Delay Time* (M)
- ☐ *Path With Minimum Effort* (M)
- ☐ *Cheapest Flights Within K Stops* (M)
- ☐ *Find the City With the Smallest Number of Neighbors at a Threshold Distance* (M)
- ☐ *Swim in Rising Water* (H)
- ☐ *Minimum Cost to Make at Least One Valid Path in a Grid* (H)

★ **Interview tip:** *Be ready to explain why Dijkstra breaks with negative edges: a node marked "final" could later get a cheaper path.*

Union-Find & Minimum Spanning Tree

Level 5 · Graphs · ≈ 3 sessions

In simple words

Union-Find (Disjoint Set Union, DSU) *keeps track of which items are in the same group. Two operations:* `find(x)` → *which group is x in? and* `union(a, b)` → *merge two groups. With two tricks it is almost O(1) per operation.*

A ***Minimum Spanning Tree (MST)*** *connects all nodes using the* ***cheapest total edge weight*** *with no cycles.*

Think of it like...

Friend circles at a party: when two people from different circles become friends, the two circles merge into one. To check if two people are connected, compare their circle leaders.

Key ideas

- ***Path compression***: while finding the root, point every node directly at the root.
- ***Union by rank/size***: attach the smaller tree under the bigger one.
- If `union(a, b)` finds they are already in the same group → that edge makes a ***cycle*** (Redundant Connection).
- Count components: start with n groups, subtract 1 on each successful union.
- DSU is great when edges arrive over time (online connectivity); BFS/DFS is simpler for a fixed graph.

Cost sheet

Algorithm	Idea	Time
DSU find / union	Path compression + union by rank	≈ $O(1)$ – $O(\alpha(n))$
Kruskal (MST)	Sort edges; add if it joins two different groups	$O(E \log E)$
Prim (MST)	Grow from one node using a min-heap	$O(E \log V)$

Java code

```
class DSU {
    int[] parent, rank;
    int groups;
    DSU(int n) {
        parent = new int[n]; rank = new int[n]; groups = n;
        for (int i = 0; i < n; i++) parent[i] = i;
    }
    int find(int x) {
        if (parent[x] != x) parent[x] = find(parent[x]);   // path compression
        return parent[x];
    }
    boolean union(int a, int b) {
        int ra = find(a), rb = find(b);
        if (ra == rb) return false;                    // already connected
        if (rank[ra] < rank[rb]) { int t = ra; ra = rb; rb = t; }
        parent[rb] = ra;
        if (rank[ra] == rank[rb]) rank[ra]++;
        groups--;
        return true;
    }
}
```

```java
// Kruskal: edges = {u, v, weight}
int kruskal(int n, int[][] edges) {
    Arrays.sort(edges, (a, b) -> Integer.compare(a[2], b[2]));
    DSU dsu = new DSU(n);
    int total = 0, used = 0;
    for (int[] e : edges) {
        if (dsu.union(e[0], e[1])) {
            total += e[2];
            if (++used == n - 1) break;
        }
    }
    return used == n - 1 ? total : -1;  // -1 if graph is disconnected
}
```

Spot it when

- → *'Are these connected?', 'number of groups / provinces'.*
- → *'Which edge creates a cycle?'*
- → *'Connect all points with minimum cost' → MST.*

Practice list

- ☐ *Number of Provinces* (M)
- ☐ *Redundant Connection* (M)
- ☐ *Accounts Merge* (M)
- ☐ *Number of Operations to Make Network Connected* (M)
- ☐ *Satisfiability of Equality Equations* (M)
- ☐ *Most Stones Removed with Same Row or Column* (M)
- ☐ *Min Cost to Connect All Points* (M)
- ☐ *Swim in Rising Water* (H)

★ ***Interview tip:*** *Write the DSU class once, cleanly, and reuse it. Interviewers like seeing a small reusable component instead of inline array juggling.*

Level 6

Dynamic Programming

Recursion with memory. The topic that separates good candidates from great ones.

In this level: Dynamic Programming Fundamentals · DP Patterns: Knapsack, LIS, LCS, Grid & More

Dynamic Programming Fundamentals

Level 6 · Dynamic Programming · ≈ 5 sessions

In simple words

*Dynamic Programming (DP) = **recursion + remembering answers**. Use it when a problem can be split into smaller subproblems that **repeat** (overlapping subproblems) and the best answer is built from best answers of smaller parts (optimal substructure).*

***Top-down (memoization)**: write the recursion, cache results. **Bottom-up (tabulation)**: fill a table from the smallest case upward, no recursion.*

> **Think of it like...**
>
> *If someone asks "what is 1+1+1+1+1?" you count 5. Add one more "+1" and you instantly say 6 — you remembered the earlier answer instead of recounting.*

Key ideas

- *Start with the **brute-force recursion**, then add a memo — this path is easy to explain and hard to get wrong.*
- *Complexity = (number of states) × (work per state).*
- *Signal words: **count the ways**, **minimum / maximum**, **is it possible**, **longest / shortest** — with choices at each step.*
- *Use `Integer` / `-1` sentinels or `Arrays.fill` for 'not computed yet' in memo tables.*
- *Many 1D DPs only depend on the previous one or two values → O(1) space.*

Cost sheet

Step	Question to answer
1. State	What does dp[i] (or dp[i][j]) mean, in plain words?
2. Recurrence	How is dp[i] built from smaller states?
3. Base cases	What are the smallest answers you know directly?
4. Order	Which order fills the table so dependencies are ready?
5. Answer	Which cell holds the final answer?
6. Optimize	Do you only need the last row / last two values? Cut space.

Java code

```
// Climbing Stairs, three ways
// 1) Top-down with memo
int[] memo;
int climbMemo(int n) {
    if (n <= 2) return n;
    if (memo[n] != 0) return memo[n];
    return memo[n] = climbMemo(n - 1) + climbMemo(n - 2);
}
// 2) Bottom-up table
int climbTable(int n) {
    if (n <= 2) return n;
    int[] dp = new int[n + 1];
    dp[1] = 1; dp[2] = 2;
    for (int i = 3; i <= n; i++) dp[i] = dp[i - 1] + dp[i - 2];
    return dp[n];
}
// 3) O(1) space
int climb(int n) {
    int a = 1, b = 1;                  // ways to reach step 0 and step 1
    for (int i = 2; i <= n; i++) { int c = a + b; a = b; b = c; }
    return b;
}
```

```
// House Robber: dp[i] = max(skip house i, rob house i + dp[i-2])
int rob(int[] nums) {
    int prev2 = 0, prev1 = 0;
    for (int x : nums) {
        int cur = Math.max(prev1, prev2 + x);
        prev2 = prev1; prev1 = cur;
    }
    return prev1;
}
```

```java
// Coin Change: fewest coins to make amount
int coinChange(int[] coins, int amount) {
    int[] dp = new int[amount + 1];
    Arrays.fill(dp, amount + 1);          // "infinity"
    dp[0] = 0;
    for (int a = 1; a <= amount; a++)
        for (int c : coins)
            if (c <= a) dp[a] = Math.min(dp[a], dp[a - c] + 1);
    return dp[amount] > amount ? -1 : dp[amount];
}
```

```java
// Word Break: can s be split into dictionary words?
boolean wordBreak(String s, List<String> dict) {
    Set<String> words = new HashSet<>(dict);
    boolean[] dp = new boolean[s.length() + 1];
    dp[0] = true;
    for (int i = 1; i <= s.length(); i++)
        for (int j = 0; j < i; j++)
            if (dp[j] && words.contains(s.substring(j, i))) { dp[i] = true; break; }
    return dp[s.length()];
}
```

Spot it when

- → *'Number of ways to...'*
- → *'Minimum cost / maximum profit'.*
- → *Greedy fails and backtracking is too slow.*

Practice list

- ☐ *Climbing Stairs* (E)
- ☐ *Min Cost Climbing Stairs* (E)
- ☐ *House Robber* (M)
- ☐ *House Robber II* (M)
- ☐ *Decode Ways* (M)
- ☐ *Coin Change* (M)
- ☐ *Word Break* (M)
- ☐ *Maximum Product Subarray* (M)
- ☐ *Perfect Squares* (M)

★ ***Interview tip:*** *Say the state definition out loud in one sentence ("dp[i] is the fewest coins to make amount i"). If you can't, you're not ready to code yet.*

DP Patterns: Knapsack, LIS, LCS, Grid & More

Level 6 · Dynamic Programming · ≈ 7 sessions

In simple words

Most interview DP problems are variations of a small set of **patterns**. Learn to recognise the pattern and the state almost writes itself.

> **Think of it like...**
>
> Like cooking: once you know how to make a basic gravy, a dozen curries are just variations.

Key ideas

- **0/1 knapsack in 1D**: loop capacity **backwards** so each item is used once. **Unbounded**: loop **forwards**.
- Coin Change II (count combinations): loop **coins outside**, amounts inside. Swapping the loops counts permutations instead.
- **LIS in O(n log n)**: keep `tails[k]` = smallest ending value of an increasing subsequence of length k+1; binary search each number into it.
- **Edit distance**: if characters match, copy the diagonal; otherwise 1 + min(insert, delete, replace).
- Interval DP fills by **length** of the interval, shortest first.
- Two-string DP tables of size (m+1) × (n+1) make the empty-prefix base case natural.

Cost sheet

Pattern	State	Classic problems
0/1 Knapsack	dp[i][w] = best using first i items with capacity w	Partition Equal Subset Sum, Target Sum
Unbounded knapsack	dp[a] = ways / min coins for amount a	Coin Change, Coin Change II
Longest Increasing Subsequence	dp[i] = LIS ending at i (or tails array)	LIS, Russian Doll Envelopes
Two strings (LCS / edit)	dp[i][j] = answer for prefixes s[0..i), t[0..j)	LCS, Edit Distance, Distinct Subsequences
Grid paths	dp[r][c] = ways / min cost to reach cell	Unique Paths, Minimum Path Sum
Interval / palindrome	dp[i][j] = answer for substring i..j (fill by length)	Longest Palindromic Subsequence, Burst Balloons
State machine	dp[i][holding / not holding / cooldown]	Stock with Cooldown, Stock with Fee
DP on trees	return {rob, skip} from each child	House Robber III
Bitmask DP	dp[mask] = best for this set of used items	Partition to K Equal Sum Subsets, TSP

Java code

```
// 0/1 Knapsack (1D): max value within capacity W
int knapsack(int[] wt, int[] val, int W) {
    int[] dp = new int[W + 1];
    for (int i = 0; i < wt.length; i++)
        for (int w = W; w >= wt[i]; w--)          // backwards!
            dp[w] = Math.max(dp[w], dp[w - wt[i]] + val[i]);
    return dp[W];
}
```

```
// Partition Equal Subset Sum = knapsack with booleans
boolean canPartition(int[] nums) {
    int sum = Arrays.stream(nums).sum();
    if (sum % 2 == 1) return false;
    boolean[] dp = new boolean[sum / 2 + 1];
    dp[0] = true;
    for (int x : nums)
        for (int s = sum / 2; s >= x; s--) dp[s] |= dp[s - x];
    return dp[sum / 2];
}
```

```
// Longest Common Subsequence
int lcs(String a, String b) {
    int m = a.length(), n = b.length();
    int[][] dp = new int[m + 1][n + 1];
    for (int i = 1; i <= m; i++)
        for (int j = 1; j <= n; j++)
            dp[i][j] = (a.charAt(i - 1) == b.charAt(j - 1))
                    ? dp[i - 1][j - 1] + 1
                    : Math.max(dp[i - 1][j], dp[i][j - 1]);
    return dp[m][n];
}
```

```
// Edit Distance
int minDistance(String a, String b) {
    int m = a.length(), n = b.length();
    int[][] dp = new int[m + 1][n + 1];
    for (int i = 0; i <= m; i++) dp[i][0] = i;
    for (int j = 0; j <= n; j++) dp[0][j] = j;
    for (int i = 1; i <= m; i++)
        for (int j = 1; j <= n; j++)
            dp[i][j] = (a.charAt(i - 1) == b.charAt(j - 1))
                    ? dp[i - 1][j - 1]
                    : 1 + Math.min(dp[i - 1][j - 1], Math.min(dp[i - 1][j], dp[i][j - 1]));
    return dp[m][n];
}
```

```java
// LIS in O(n log n)
int lengthOfLIS(int[] nums) {
    int[] tails = new int[nums.length];
    int size = 0;
    for (int x : nums) {
        int lo = 0, hi = size;
        while (lo < hi) {                    // first tail >= x
            int mid = (lo + hi) >>> 1;
            if (tails[mid] < x) lo = mid + 1; else hi = mid;
        }
        tails[lo] = x;
        if (lo == size) size++;
    }
    return size;
}
```

```java
// Unique Paths in a grid (1D rolling row)
int uniquePaths(int m, int n) {
    int[] row = new int[n];
    Arrays.fill(row, 1);
    for (int r = 1; r < m; r++)
        for (int c = 1; c < n; c++) row[c] += row[c - 1];
    return row[n - 1];
}
```

Spot it when

- ➜ *'Subset with sum...', 'split into two equal parts' → knapsack.*
- ➜ *'Two strings... minimum operations / common...' → 2D string DP.*
- ➜ *'Increasing subsequence' → LIS.*
- ➜ *'Burst / merge / cut in best order' → interval DP.*

Practice list

- ☐ *Unique Paths* (M)
- ☐ *Minimum Path Sum* (M)
- ☐ *Partition Equal Subset Sum* (M)
- ☐ *Target Sum* (M)
- ☐ *Coin Change II* (M)
- ☐ *Longest Increasing Subsequence* (M)
- ☐ *Longest Common Subsequence* (M)
- ☐ *Edit Distance* (M)
- ☐ *Longest Palindromic Subsequence* (M)
- ☐ *Best Time to Buy and Sell Stock with Cooldown* (M)
- ☐ *House Robber III* (M)
- ☐ *Interleaving String* (M)
- ☐ *Partition to K Equal Sum Subsets* (M)
- ☐ *Distinct Subsequences* (H)
- ☐ *Burst Balloons* (H)
- ☐ *Regular Expression Matching* (H)

★ ***Interview tip:*** *After a correct 2D DP, offer the space optimisation ("each row only needs the previous row, so O(n) space"). It's an easy extra point.*

Level 7

Advanced & Design

Stretch topics for hard rounds, plus the design-a-class questions experienced Java developers get.

In this level: Segment Tree & Fenwick Tree · Advanced String Algorithms · Advanced Graph Topics · Design Data Structures (LRU & friends)

Segment Tree & Fenwick Tree

Level 7 · Advanced & Design · ≈ 3 sessions

In simple words

Prefix sums answer range queries in O(1) but an update costs O(n). When you need **both range queries and updates** quickly, use:

Fenwick Tree (Binary Indexed Tree) — short code, handles prefix sums with point updates in O(log n).
Segment Tree — more general: range sum, min, max, GCD; with **lazy propagation** it also handles range updates.

> **Think of it like...**
>
> A company's sales report: each manager keeps the total for their team, so the CEO can answer "sales for regions 3 to 7" by asking a few managers instead of every salesperson.

Key ideas

- Fenwick uses 1-based indexes and the lowest-set-bit trick `i & -i`.
- Segment tree node covers a range; children split it in half; the root covers everything.
- Counting problems ('how many smaller elements to the right') use a Fenwick tree over **compressed** values.
- These are less common in typical MNC interviews — learn them after everything else feels comfortable.

Cost sheet

Operation	Fenwick	Segment tree
Build	O(n log n) (or O(n))	O(n)
Point update	O(log n)	O(log n)
Range query	O(log n) (sums)	O(log n) (sum/min/max…)
Range update	with tricks	O(log n) with lazy propagation
Space	O(n)	O(4n)

Java code

```
// Fenwick Tree (1-based)
class Fenwick {
   int[] tree;
   Fenwick(int n) { tree = new int[n + 1]; }
   void add(int i, int delta) {              // a[i] += delta
      for (; i < tree.length; i += i & -i) tree[i] += delta;
   }
   int prefix(int i) {                       // a[1] + ... + a[i]
      int s = 0;
      for (; i > 0; i -= i & -i) s += tree[i];
      return s;
   }
   int range(int l, int r) { return prefix(r) - prefix(l - 1); }
}
```

```java
// Segment Tree for range sum
class SegmentTree {
    int n; int[] tree;
    SegmentTree(int[] a) { n = a.length; tree = new int[4 * n]; build(a, 1, 0, n - 1); }
    void build(int[] a, int node, int l, int r) {
        if (l == r) { tree[node] = a[l]; return; }
        int m = (l + r) / 2;
        build(a, 2 * node, l, m);
        build(a, 2 * node + 1, m + 1, r);
        tree[node] = tree[2 * node] + tree[2 * node + 1];
    }
    void update(int node, int l, int r, int idx, int val) {
        if (l == r) { tree[node] = val; return; }
        int m = (l + r) / 2;
        if (idx <= m) update(2 * node, l, m, idx, val);
        else          update(2 * node + 1, m + 1, r, idx, val);
        tree[node] = tree[2 * node] + tree[2 * node + 1];
    }
    int query(int node, int l, int r, int ql, int qr) {
        if (qr < l || r < ql) return 0;             // no overlap
        if (ql <= l && r <= qr) return tree[node];    // full overlap
        int m = (l + r) / 2;
        return query(2 * node, l, m, ql, qr) + query(2 * node + 1, m + 1, r, ql, qr);
    }
}
```

Spot it when

- → *'Range sum / min / max' **and** 'update index i', many times.*
- → *'Count of smaller numbers after self', 'reverse pairs'.*

Practice list

- ☐ *Range Sum Query - Mutable* (M)
- ☐ *Count of Smaller Numbers After Self* (H)
- ☐ *Reverse Pairs* (H)
- ☐ *Count of Range Sum* (H)
- ☐ *My Calendar III* (H)
- ☐ *Falling Squares* (H)

★ ***Interview tip:*** *If short on time in an interview, a Fenwick tree is ~15 lines and covers most "update + range sum" questions.*

Advanced String Algorithms

Level 7 · Advanced & Design · ≈ 2 sessions

In simple words

*Finding a pattern of length m inside a text of length n takes O(n·m) the naive way. These algorithms do it in about **O(n + m)** by reusing information from earlier comparisons.*

Think of it like...

Reading a long document for a phrase: when a partial match fails, a smart reader doesn't restart from scratch — they remember how much of the phrase they've already seen.

Key ideas

- *KMP's LPS array also solves 'shortest palindrome' and 'repeated substring pattern'.*
- *Rolling hash:* `hash = (hash × base + c) mod M` *; remove the leftmost char by subtracting its weight. Watch for collisions — verify on match.*
- *Rolling hash + binary search finds the 'longest duplicate substring'.*
- *In interviews,* `indexOf` *is fine unless they explicitly ask you to implement matching.*

Cost sheet

Algorithm	Idea	Time
KMP	LPS array: longest proper prefix that is also a suffix	O(n + m)
Rabin-Karp	Rolling hash of each window; compare only on hash match	O(n + m) average
Z-algorithm	Z[i] = length of match between s and s[i..]	O(n + m)
Manacher	All palindromic substrings	O(n)

Java code

```
// KMP: first index of pattern in text, or -1
int strStr(String text, String pat) {
  if (pat.isEmpty()) return 0;
  int[] lps = buildLps(pat);
  int j = 0;                          // chars matched so far
  for (int i = 0; i < text.length(); i++) {
    while (j > 0 && text.charAt(i) != pat.charAt(j)) j = lps[j - 1];
    if (text.charAt(i) == pat.charAt(j)) j++;
    if (j == pat.length()) return i - j + 1;
  }
  return -1;
}
int[] buildLps(String p) {
  int[] lps = new int[p.length()];
  int len = 0;
  for (int i = 1; i < p.length(); i++) {
    while (len > 0 && p.charAt(i) != p.charAt(len)) len = lps[len - 1];
    if (p.charAt(i) == p.charAt(len)) len++;
    lps[i] = len;
  }
  return lps;
}
```

Spot it when

- → 'Find pattern in text' with large inputs.
- → 'Repeated substring', 'longest prefix that is also a suffix'.

Practice list

- ☐ Find the Index of the First Occurrence in a String (E)
- ☐ Repeated Substring Pattern (E)
- ☐ Repeated DNA Sequences (M)
- ☐ Longest Happy Prefix (H)
- ☐ Shortest Palindrome (H)
- ☐ Longest Duplicate Substring (H)

★ **Interview tip:** Know KMP well enough to explain the LPS array with a small example like "ababaca" — that is usually what's asked.

Advanced Graph Topics

Level 7 · Advanced & Design · ≈ 2 sessions

In simple words

A few graph ideas appear in harder rounds: finding ***critical edges*** whose removal disconnects a network, ***strongly connected components*** in directed graphs, and ***Eulerian paths*** that use every edge exactly once.

Think of it like...

In a telecom network, a "bridge" is a single link that, if it fails, cuts a region off — exactly what Tarjan's algorithm finds.

Key ideas

- ✱ **`low[u]`** = earliest discovery time reachable from u's subtree using one back edge. Edge (u, v) is a bridge if `low[v] > disc[u]`.
- ✱ Kosaraju: DFS to get finish order → reverse all edges → DFS in reverse finish order; each DFS tree is one SCC.
- ✱ Reconstruct Itinerary is an Eulerian path: DFS with a min-heap of destinations, add airports on the way back.
- ✱ Safe states = nodes not reachable into a cycle (reverse graph + topo sort).

Cost sheet

Topic	Algorithm	Time
Bridges & articulation points	Tarjan (discovery time + low-link)	$O(V + E)$
Strongly connected components	Kosaraju (2 DFS) or Tarjan	$O(V + E)$
Eulerian path	Hierholzer	$O(E)$
Max flow (awareness)	Ford-Fulkerson / Edmonds-Karp	$O(V \cdot E^2)$
Minimum height trees	Peel leaves layer by layer (topo-like)	$O(V)$

Java code

```
// Tarjan: find all bridges (critical connections)
int timer = 0;
int[] disc, low;
List<List<Integer>> bridges = new ArrayList<>();
```

```
List<List<Integer>> criticalConnections(int n, List<List<Integer>> conns) {
    List<List<Integer>> g = new ArrayList<>();
    for (int i = 0; i < n; i++) g.add(new ArrayList<>());
    for (List<Integer> e : conns) { g.get(e.get(0)).add(e.get(1)); g.get(e.get(1)).add(e.get(0)); }
    disc = new int[n]; low = new int[n];
    Arrays.fill(disc, -1);
    for (int i = 0; i < n; i++) if (disc[i] == -1) dfs(i, -1, g);
    return bridges;
}
void dfs(int u, int parent, List<List<Integer>> g) {
    disc[u] = low[u] = timer++;
    for (int v : g.get(u)) {
        if (v == parent) continue;
        if (disc[v] == -1) {
            dfs(v, u, g);
            low[u] = Math.min(low[u], low[v]);
            if (low[v] > disc[u]) bridges.add(List.of(u, v));  // no back edge around (u, v)
        } else {
            low[u] = Math.min(low[u], disc[v]);              // back edge
        }
    }
}
```

Spot it when

- 'Critical connection', 'single point of failure'.
- 'Groups where everyone can reach everyone' (directed).
- 'Use every ticket / edge exactly once'.

Practice list

- [ ] Find Eventual Safe States (M)
- [ ] Minimum Height Trees (M)
- [ ] Critical Connections in a Network (H)
- [ ] Reconstruct Itinerary (H)
- [ ] Bus Routes (H)
- [ ] Parallel Courses III (H)

★ **Interview tip:** These are "stretch" topics. Cover them only after Levels 0–6 feel solid.

Design Data Structures (LRU & friends)

Level 7 · Advanced & Design · ≈ 3 sessions

In simple words

"Design" coding questions ask you to build a small class that combines basic structures to hit strict complexity targets — usually **O(1) per operation**. They're very common for experienced backend developers because they mirror real caching and rate-limiting work.

Think of it like...

A desk with limited space: you keep recently used files on top and throw out the one you haven't touched for the longest time — that's an LRU cache.

Key ideas

- LRU: the map gives O(1) lookup; the linked list gives O(1) 'move to front' and 'remove oldest'.
- Use **dummy head and tail** nodes in the linked list so you never check for null.
- Java shortcut: `LinkedHashMap` with `accessOrder = true` and `removeEldestEntry`. Offer it, but be ready to write the manual version.
- Thread safety question? Mention `ConcurrentHashMap`, locks, or Caffeine/Guava caches in real services.

Cost sheet

Design	Building blocks	Target
LRU Cache	HashMap + doubly linked list (or LinkedHashMap)	get / put O(1)
LFU Cache	HashMap + map of frequency → linked list	O(1)
Insert / Delete / GetRandom	ArrayList + HashMap (swap with last on delete)	O(1)
Time-based key-value store	HashMap + sorted list + binary search	set O(1), get O(log n)
Hit counter / rate limiter	Queue of timestamps or fixed buckets	O(1) amortized
Consistent hashing (system design link)	TreeMap ring + ceilingKey	O(log n)

Java code

```
// LRU Cache: HashMap + doubly linked list
class LRUCache {
  private static class Node {
    int key, val; Node prev, next;
    Node(int k, int v) { key = k; val = v; }
  }
  private final int capacity;
  private final Map<Integer, Node> map = new HashMap<>();
  private final Node head = new Node(0, 0), tail = new Node(0, 0);  // dummies
```

```
    LRUCache(int capacity) {
        this.capacity = capacity;
        head.next = tail; tail.prev = head;
    }
    public int get(int key) {
        Node n = map.get(key);
        if (n == null) return -1;
        remove(n); addFront(n);              // mark as most recent
        return n.val;
    }
    public void put(int key, int value) {
        Node n = map.get(key);
        if (n != null) { n.val = value; remove(n); addFront(n); return; }
        if (map.size() == capacity) {
            Node lru = tail.prev;            // least recently used
            remove(lru); map.remove(lru.key);
        }
        n = new Node(key, value);
        map.put(key, n); addFront(n);
    }
    private void remove(Node n)  { n.prev.next = n.next; n.next.prev = n.prev; }
    private void addFront(Node n) {
        n.next = head.next; n.prev = head;
        head.next.prev = n; head.next = n;
    }
}
```

```
// The quick Java version
class LRUQuick<K, V> extends LinkedHashMap<K, V> {
    private final int cap;
    LRUQuick(int cap) { super(16, 0.75f, true); this.cap = cap; }  // accessOrder = true
    @Override protected boolean removeEldestEntry(Map.Entry<K, V> e) { return size() > cap; }
}
```

```java
// Insert Delete GetRandom O(1)
class RandomizedSet {
  private final List<Integer> list = new ArrayList<>();
  private final Map<Integer, Integer> index = new HashMap<>();
  private final Random rnd = new Random();
  public boolean insert(int v) {
    if (index.containsKey(v)) return false;
    index.put(v, list.size()); list.add(v);
    return true;
  }
  public boolean remove(int v) {
    Integer i = index.get(v);
    if (i == null) return false;
    int last = list.get(list.size() - 1);
    list.set(i, last); index.put(last, i);     // move last into the hole
    list.remove(list.size() - 1); index.remove(v);
    return true;
  }
  public int getRandom() { return list.get(rnd.nextInt(list.size())); }
}
```

Spot it when

- → *'Design a class that supports... in O(1)'.*
- → *'Cache', 'eviction', 'recent', 'rate limit', 'timestamp'.*

Practice list

- ☐ *LRU Cache* (M)
- ☐ *Insert Delete GetRandom O(1)* (M)
- ☐ *Time Based Key-Value Store* (M)
- ☐ *Design Twitter* (M)
- ☐ *Design Hit Counter* (M)
- ☐ *LFU Cache* (H)
- ☐ *All O`one Data Structure* (H)

★ **Interview tip:** *LRU Cache is one of the most-asked questions for experienced Java developers. Practise writing the manual version in under 15 minutes.*

Pattern finder

Read the problem → find the clue on the left → try the technique on the right.

If the problem says...	Try
Sorted array, find a pair or triplet	**Two pointers**
Contiguous subarray / substring, longest or shortest	**Sliding window**
Count subarrays with sum K (negatives allowed)	**Prefix sum + HashMap**
Have I seen this before? Find a duplicate or complement	**HashMap / HashSet**
K largest, K most frequent, K closest	**Heap of size K**
Median of a stream	**Two heaps**
Next greater / smaller element, span, histogram	**Monotonic stack**
Maximum of every window of size k	**Monotonic deque**
Minimize the maximum / maximize the minimum	**Binary search on the answer**
Sorted or rotated array, O(log n) required	**Binary search**
Overlapping ranges, meetings, bookings	**Sort + merge, or sweep line**
Linked list middle, cycle, k-th from end	**Fast & slow pointers**
Matching brackets, undo, nested decoding	**Stack**
Tree height, path, subtree answer	**DFS returning values from children**
Tree level by level, right-side view	**BFS with a queue**
Words sharing prefixes, autocomplete	**Trie**
Return all subsets / permutations / combinations	**Backtracking**
Pick the earliest ending, farthest reach	**Greedy (after sorting)**
Minimum steps on a grid or unweighted graph	**BFS**
Islands, regions, connected parts	**DFS / BFS or Union-Find**
Prerequisites, dependencies, build order	**Topological sort**
Cheapest / fastest path with weights	**Dijkstra (Bellman-Ford if negative)**
Are these connected? Which edge makes a cycle?	**Union-Find**
Connect all points at minimum cost	**Minimum spanning tree**
Count the ways, min cost, max profit with choices	**Dynamic programming**
Two strings: common part, edit operations	**2D string DP**
Subset with a given sum, split into equal halves	**0/1 knapsack DP**
Appears once, no extra space, subsets of $n \leq 20$	**Bit manipulation**
Range sum with updates	**Fenwick / segment tree**
Design a class with O(1) get / put / evict	**HashMap + doubly linked list**

Complexity at a glance

Structure	Access	Search	Insert	Delete
Array	$O(1)$	$O(n)$	$O(n)$	$O(n)$
ArrayList (at end)	$O(1)$	$O(n)$	$O(1)^*$	$O(1)$
Linked list (at head)	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Stack / Queue (ArrayDeque)	top $O(1)$	$O(n)$	$O(1)$	$O(1)$
HashMap / HashSet	—	$O(1)$ avg	$O(1)$ avg	$O(1)$ avg
TreeMap / balanced BST	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Heap (PriorityQueue)	top $O(1)$	$O(n)$	$O(\log n)$	top $O(\log n)$
Trie	—	$O(L)$	$O(L)$	$O(L)$
Union-Find	—	find $\approx O(1)$	union $\approx O(1)$	—
Segment / Fenwick tree	—	range $O(\log n)$	update $O(\log n)$	—

* amortized · L = word length

Sorting cheat line

Sort	Average	Worst	Space	Stable
Insertion	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Merge	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes
Quick	$O(n \log n)$	$O(n^2)$	$O(\log n)$	No
Heap	$O(n \log n)$	$O(n \log n)$	$O(1)$	No
Counting	$O(n + k)$	$O(n + k)$	$O(k)$	Yes

Input size → target complexity

n up to	Aim for
10	$O(n!)$
20	$O(2^n)$
500	$O(n^3)$
10^4	$O(n^2)$
10^6	$O(n \log n)$ or $O(n)$
10^{9}+	$O(\log n)$ or $O(1)$

Edge-case checklist

- [ ] *Empty input, one element, two elements*
- [ ] *All elements equal; many duplicates*
- [ ] *Negative numbers and zero*
- [ ] *Very large values → int overflow, use long*
- [ ] *Already sorted and reverse sorted input*
- [ ] *null head / root; single-node tree; skewed tree*
- [ ] *Disconnected graph; graph with a cycle; self-loops*
- [ ] *Target not present; answer at the first or last index*

Where to practise

- → **LeetCode** — *every problem in these notes (a few, like Meeting Rooms and Alien Dictionary, are Premium)*
- → **NeetCode 150** — *list grouped by pattern, with free versions of Premium problems*
- → **GeeksforGeeks** — *theory and company-wise questions*
- → **VisuAlgo** — *animations of sorting, trees and graphs*

One topic at a time. You've got this! ✓