CODING, SYSTEM DESIGN, AND AI, IN ONE PLACE

Interview *Prep*

Three parts in one book. Competitive programming patterns in plain English with working Python and Java. System design from first principles through real products like WhatsApp, Amazon, and YouTube. And a hands on tour of modern AI, from how models work to RAG, agents, memory, MCP, and skills.

Competitive Programming · System Design · AI Engineering · Diagrams · Real world examples

Contents

INTERVIEW PREP

Competitive Programming

System Design

AI Engineering

PART ONE

Competitive Programming

The patterns behind almost every coding problem, from arrays and two pointers to graphs and dynamic programming. Each one is explained in plain English, drawn as a picture, and written out in both Python and Java, with the problem types it unlocks and the traps that catch people.

FOUNDATIONS

00 How to use this book

Solving LeetCode problems is not about being a genius. It is a skill, and like any skill it breaks down into a small number of patterns that show up again and again. Once you can recognize the pattern, most problems become variations of something you have already done.

This book is organized around those patterns, not around random problems. Each chapter follows the same shape so you always know what you are getting:

- **The idea in plain English.** What the pattern actually is, without jargon.
- **When to reach for it.** The signals in a problem statement that tell you "this is the one."
- **A picture.** Almost every pattern is easier to see than to read.
- **A template.** The skeleton of code you can adapt, so you are not starting from a blank page.
- **Worked examples** in both Python and Java, with the thinking spelled out.
- **Tricks and traps** that save you time or stop you from failing on edge cases.

> HOW TO ACTUALLY LEARN THIS
>
> Reading is not practice. The loop that works is simple: read the pattern here, then go solve three or four problems that use it before moving on. Struggle for about twenty minutes before looking at a hint. When you get stuck and read the answer, close it and rewrite the solution from memory. If you cannot rewrite it, you have not learned it yet.

You do not need to read this front to back. If you already know arrays and want trees, jump there. But the Foundations chapters, especially how to approach a problem and how to think about time complexity, pay off in every single chapter after them, so start there if you are newer to this.

01 Approaching any problem

Most people fail a problem not because the code is hard, but because they start typing before they understand what they are solving. Here is a repeatable process that works whether the problem is easy or brutal.

The five steps

1. Understand the problem completely

Read it twice. Write down, in your own words, what the input is and what the output should be. Then invent one small example by hand and solve it on paper. If you cannot solve a tiny case by hand, you definitely cannot code it. Pay attention to the constraints (the numbers like "1 <= n <= 10^5"), because they are secretly telling you which approach is allowed.

> CONSTRAINTS ARE A CHEAT SHEET
>
> The size of the input hints at the target time complexity. If n can be up to 10^5 or more, an O(n^2) solution will time out, so you need O(n log n) or O(n). If n is tiny, say up to 20, an exponential solution like backtracking is probably expected. Read the constraints before you decide on an approach, not after.

2. Find a brute force answer first

Do not chase the clever solution immediately. Ask "what is the dumbest thing that would work?" Often it is: try every possibility. Getting brute force clear in your head does two things. It confirms you understand the problem, and it gives you something to optimize. Many interviewers are happy if you state the brute force out loud even if you never code it.

3. Look for the bottleneck and the pattern

Ask where the brute force wastes work. Are you recomputing the same thing? A hash map or memoization might help. Are you scanning a sorted array repeatedly? Binary search or two pointers might help. Are you exploring choices step by step? That is backtracking or dynamic programming. This is where knowing the patterns in this book turns a dead end into a plan.

4. Code it in small pieces

Write the structure first, then fill it in. Handle the normal case, then come back for the edge cases. Keep variable names honest, so `left` and `right` instead of `a` and `b`. Clean code has fewer bugs.

5. Test with your own cases

Before you hit submit, run the example you made in step one in your head, line by line. Then think about the nasty inputs: empty input, one element, all elements the same, the largest allowed size, negative numbers. Most wrong answers on LeetCode are edge cases, not logic mistakes.

THE EDGE CASES THAT BITE EVERYONE

Empty array or string. A single element. Duplicates. Negative numbers and zero. Integer overflow in Java (an `int` maxes near 2.1 billion, so use `long` when summing or multiplying). The pointer at the very start or very end of an array. Check these every time.

A quick way to guess the pattern

What you see in the problem	Pattern to try first
"Find two numbers that...", "have you seen this before"	Hash map
Sorted array, or "pair that sums to X"	Two pointers
"Longest / shortest subarray or substring"	Sliding window
"Search in sorted", or "minimum value that works"	Binary search
Matching brackets, "next greater element", undo	Stack
Tree or grid, "levels", "shortest path unweighted"	BFS
"All combinations / permutations / subsets"	Backtracking
"Number of ways", "min or max cost", overlapping subproblems	Dynamic programming
"Top K", "K largest / smallest", running median	Heap
Connected groups, islands, "is A reachable from B"	Union-Find or DFS/BFS

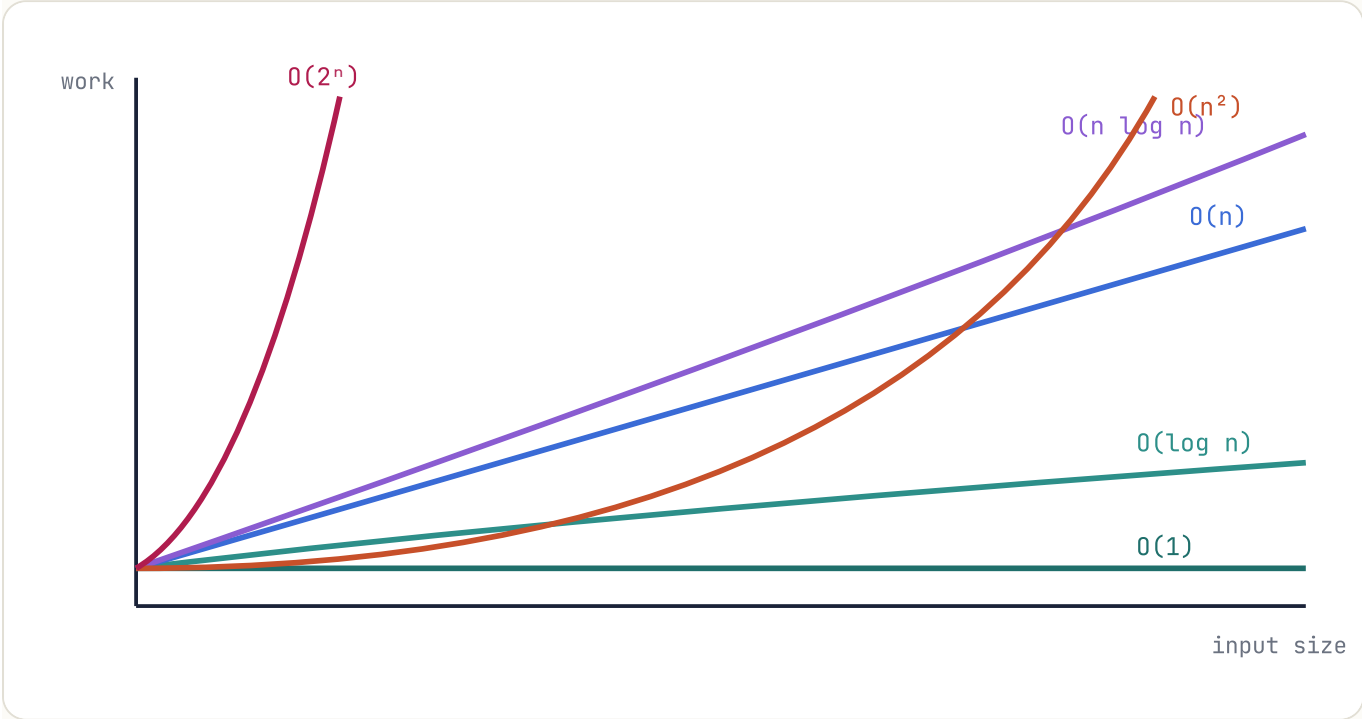
Keep this table close while you practice. Pattern recognition is the whole game, and it becomes automatic faster than you expect.

02

Big O and complexity

Big O is just a way to describe how the amount of work grows as the input grows. It answers one question: if I make the input ten times bigger, does my solution get ten times slower, a hundred times slower, or barely slower at all?

You do not need heavy math for this. You need to look at your loops. A single loop over n items is O(n). A loop inside a loop is usually O(n^2). Cutting the problem in half each step is O(log n). That is 90 percent of what you will use.



The same input, wildly different growth. Aim for the flatter curves.

The common complexities, from best to worst

Notation	Name	What it looks like in code
`O(1)`	Constant	Direct access, a hash lookup, arithmetic. Size does not matter.
`O(log n)`	Logarithmic	Halving the search space each step. Binary search.
`O(n)`	Linear	One pass through the data.
`O(n log n)`	Linearithmic	Good sorting, or a loop that does a log-n operation each time.
`O(n²)`	Quadratic	Nested loops over the same data. Fine for small n, slow for big n.
`O(2ⁿ)`	Exponential	Trying every subset or every path. Only okay when n is tiny.

THE RULES FOR READING BIG O

Drop the constants: O(2n) is just O(n). Keep only the biggest term: $O(n^2 + n)$ is $O(n^2)$. Sequential loops add, so two separate loops are O(n) + O(n) = O(n). Nested loops multiply, so a loop inside a loop is $O(n \times n) = O(n^2)$.

Do not forget space

Time is how long it runs. Space is how much extra memory it uses. A hash map holding n items is O(n) space. Recursion also costs space, because each call sits on the call stack until it returns, so recursion that goes n deep uses O(n) space even if you never allocated an array. Interviewers care about both.

A worked example: counting the cost

Look at these two functions that both check for a duplicate. One is $O(n^2)$, one is O(n). Same result, very different speed.

PYTHON

```python
# O(n^2): for each element, scan the rest. Slow.
def has_duplicate_slow(nums):
    for i in range(len(nums)):
        for j in range(i + 1, len(nums)):
            if nums[i] == nums[j]:
                return True
    return False

# O(n): remember what you have seen in a set. One pass.
def has_duplicate_fast(nums):
    seen = set()
    for x in nums:
        if x in seen:
            return True
        seen.add(x)
    return False
```

JAVA

```java
// O(n^2): for each element, scan the rest. Slow.
boolean hasDuplicateSlow(int[] nums) {
    for (int i = 0; i < nums.length; i++) {
        for (int j = i + 1; j < nums.length; j++) {
            if (nums[i] == nums[j]) return true;
        }
    }
    return false;
}

// O(n): remember what you have seen in a set. One pass.
boolean hasDuplicateFast(int[] nums) {
    Set<Integer> seen = new HashSet<>();
    for (int x : nums) {
        if (seen.contains(x)) return true;
        seen.add(x);
    }
    return false;
}
```

The fast version trades a little memory (the set) for a lot of speed. That trade, spending memory to save time, is the single most common optimization in this whole book. Keep an eye out for it.

GOING DEEPER

The complexity classes you actually meet, and what input sizes they survive

A rough rule of thumb: a modern judge does somewhere around 100 million simple operations per second. So the size of the input quietly tells you which complexity you are allowed to have. This table is worth burning into memory, because it turns "is my idea fast enough" into a two second check.

Complexity	What it looks like, and the input size it handles
O(1)	Constant work, like a hash lookup or arithmetic. Handles any size.
O(log n)	Binary search, balanced tree operations. Handles enormous n, even 10^18.
O(n)	A single pass. Comfortable up to about 10^7 or 10^8.
O(n log n)	Sorting, or a pass with a heap. Comfortable up to about 10^6.
O(n^2)	Nested loops over the input. Fine only up to a few thousand (about 10^4).
O(n^3)	Triple loops, some interval DP. Fine only up to a few hundred.
O(2^n)	Trying every subset, plain backtracking. Only tiny n, up to about 20 to 25.
O(n!)	Every permutation. Only very tiny n, up to about 10 or 11.

READ THE CEILING BACKWARDS

If the problem says n can be 100000, you are being told to find an O(n) or O(n log n) solution, because O(n^2) would be ten billion operations and time out. If n is at most 20, you are being invited to do something exponential like backtracking. The constraints are a hint, not just a limit.

A word on amortized and average cost

Some operations are usually cheap but occasionally expensive, and we care about the average over many calls. Appending to a dynamic array is O(1) amortized: most appends are instant, and the rare resize that copies everything is spread thin across all the cheap ones. Hash map lookups are O(1) on average but O(n) in a pathological worst case where everything collides. For interviews and contests, the average and amortized numbers are what you plan around, but it is worth knowing the worst case exists.

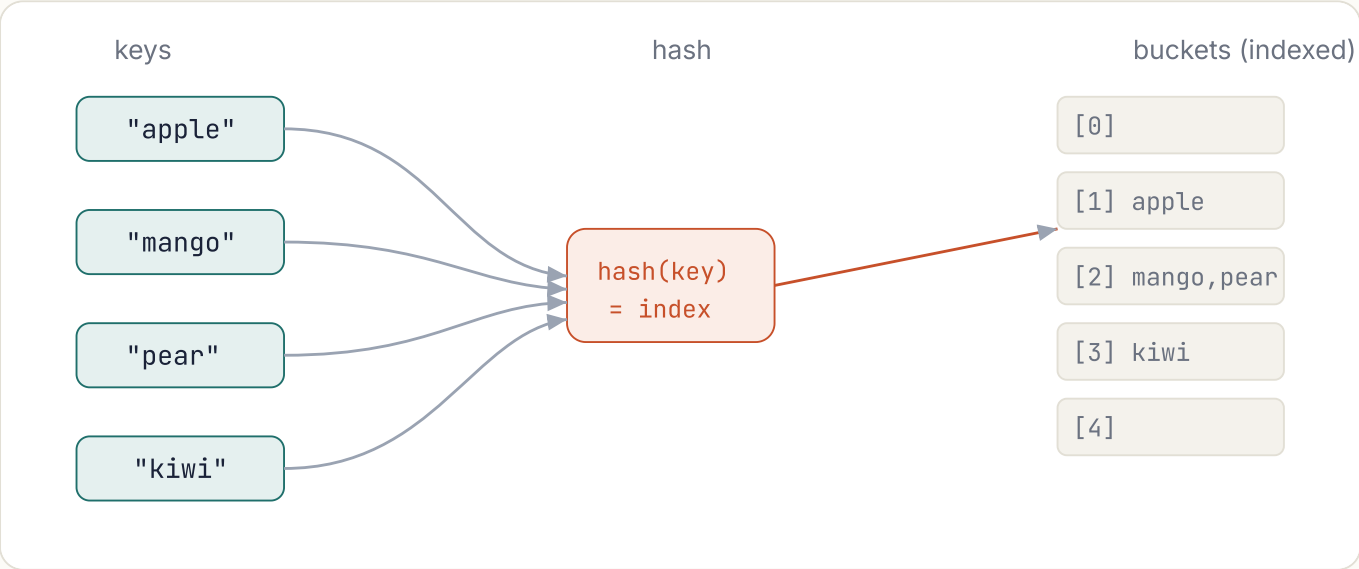
CORE PATTERNS

03 Arrays and hashing

Arrays are just a row of boxes with numbered positions. Hashing is the trick of storing things by a key so you can find them again instantly. Put them together and you can solve a huge slice of all problems.

Why hashing is a superpower

A hash map (called `dict` in Python and `HashMap` in Java) lets you answer "have I seen this before, and where?" in roughly constant time. Instead of scanning the whole array again and again, you remember what you have seen as you go. A hash set is the same idea when you only care about presence, not a value attached to it.



A key is turned into a bucket index, so lookup does not depend on how many items you stored.

REACH FOR HASHING WHEN YOU SEE

"Find a pair / complement", "count how many times each thing appears", "have I seen this value", "group things that are the same in some way", or "check for duplicates". Anytime the brute force is a nested loop that keeps re-scanning, a hash map usually flattens it to a single pass.

Worked example: Two Sum

Problem. Given an array and a target, return the indices of the two numbers that add up to the target.

Brute force. Check every pair. That is two nested loops, $O(n^2)$.

The insight. For each number `x`, the partner you need is `target - x`. If you remember every number you have already passed in a map from value to its index, then for each new number you can ask the map "have I already seen your partner?" in one step. That turns $O(n^2)$ into $O(n)$.

PYTHON

```python
def two_sum(nums, target):
    seen = {}                        # value → index we saw it at
    for i, x in enumerate(nums):
        need = target - x            # the partner that completes the pair
        if need in seen:             # have we already passed the partner?
            return [seen[need], i]
        seen[x] = i                  # remember this number for later
    return []                        # no pair found
```

JAVA

```java
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];               // partner we need
        if (seen.containsKey(need)) {              // seen it already?
            return new int[]{ seen.get(need), i };
        }
        seen.put(nums[i], i);                      // remember this one
    }
    return new int[]{};                            // no pair found
}
```

Notice we store each number only after checking, so we never pair a number with itself. That ordering matters.

The two other array moves you will use constantly

Frequency counting

Count how many times each item appears. This solves anagrams, majority element, "first unique character", and many more. In Python, `collections.Counter` does it in one line. In Java, use a `HashMap` with `getOrDefault`.

PYTHON

```python
from collections import Counter

def is_anagram(s, t):
    # Two strings are anagrams if they use the exact same letters.
    return Counter(s) == Counter(t)
```

JAVA

```java
boolean isAnagram(String s, String t) {
    if (s.length() != t.length()) return false;
    int[] count = new int[26];           // one slot per lowercase letter
    for (int i = 0; i < s.length(); i++) {
        count[s.charAt(i) - 'a']++;      // add for s
        count[t.charAt(i) - 'a']--;      // remove for t
    }
    for (int c : count) if (c != 0) return false;
    return true;                          // all balanced → anagram
}
```

LETTERS MAP TO NUMBERS

When a problem is only lowercase English letters, you can skip the hash map entirely and use an array of size 26. The index is `c - 'a'`, which turns 'a' into 0, 'b' into 1, and so on. An array is faster than a map and shows you understand the constraint.

Grouping

Put items into buckets keyed by something they share. Classic example: group anagrams together. The shared key is the sorted version of each word, since all anagrams sort to the same string.

PYTHON

```python
from collections import defaultdict

def group_anagrams(words):
    groups = defaultdict(list)
    for w in words:
        key = "".join(sorted(w))   # anagrams share the same sorted key
        groups[key].append(w)
    return list(groups.values())
```

JAVA

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

WATCH OUT

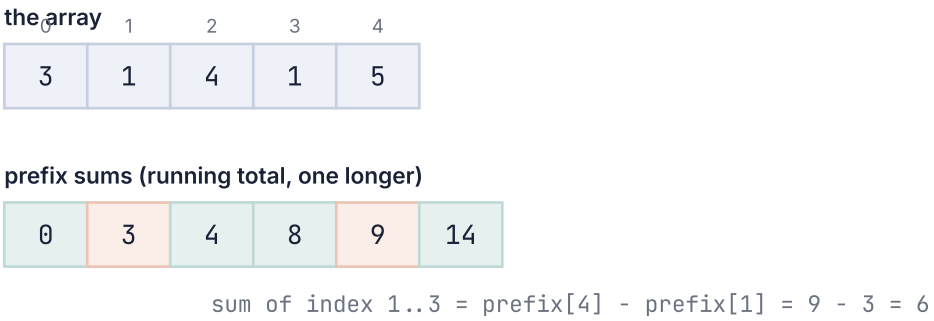
Hash lookups are "on average" O(1), not guaranteed. Also, order is not preserved in a plain Java `HashMap`; if you need insertion order, use `LinkedHashMap`. In Python, regular `dict` keeps insertion order, which is handy.

DEEPER INTUITION

Why hashing and prefix sums work

Both tricks remove repeated work. A hash map turns "have I seen this value" from a full rescan into a single O(1) lookup, so a nested loop collapses to one pass. Prefix sums precompute every running

total once, so the sum of any range is just the difference of two stored totals, again O(1) instead of re adding the range each time.



A prefix array stores running totals, so any range sum becomes one subtraction.

Another worked example: Subarray Sum Equals K

Count how many contiguous subarrays add up to k. Keep a running total and a map of how many times each total has appeared. If the current total minus k has been seen before, every one of those earlier positions starts a subarray that sums to k. One pass, O(n).

PYTHON

```python
def subarray_sum(nums, k):
    count = 0
    running = 0
    seen = {0: 1}                       # one empty prefix with sum 0
    for x in nums:
        running += x
        count += seen.get(running - k, 0)   # earlier prefixes that hit k
        seen[running] = seen.get(running, 0) + 1
    return count
```

JAVA

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

GOING DEEPER

What kinds of problems this solves

USE THIS PATTERN FOR

Anything about seeing a value before, counting how often things appear, pairing elements toward a target, grouping by a shared key, or answering range-sum questions quickly. Whenever a brute force keeps rescanning the array, a hash structure or a prefix sum usually removes the rescanning.

Problem type	Classic examples
Have I seen this	Contains Duplicate, Two Sum, Longest Consecutive Sequence, Intersection of Two Arrays
Frequency counting	Valid Anagram, Group Anagrams, Top K Frequent Elements, Majority Element, First Unique Character
Prefix sums	Subarray Sum Equals K, Range Sum Query, Product of Array Except Self, Contiguous Array
Grouping by a key	Group Anagrams, Sort Characters By Frequency, Find All Duplicates in an Array

The algorithm, in a bit more detail

A hash map trades memory for speed. Instead of scanning the array again to answer "have I seen x", you record every value you pass in a map, so the next lookup is a single O(1) step. That one move turns many O(n squared) brute forces into O(n).

Prefix sums are the other core idea. If you precompute a running total, then the sum of any range from i to j is just `prefix[j] - prefix[i-1]`, computed in O(1). Combine a prefix sum with a hash map and you can count subarrays that hit a target sum in a single pass, which is exactly how "Subarray Sum Equals K" works.

Variations you will run into

The three shapes you will use constantly are a **hash set** when you only care whether something exists, a **hash map** when you need to remember a value like an index or a count, and a **running prefix sum plus a map** when the question is about contiguous ranges. Knowing which of the three fits is most of the work.

Edge cases and gotchas

- Store a number in the map only after you check for its partner, otherwise a value can wrongly pair with itself.
- Hash maps do not keep insertion or sorted order, so never rely on iteration order for correctness.

- In Java use getOrDefault or merge for counting, and remember keys must be objects, so autoboxing of int to Integer is happening under the hood.
- Lookups are O(1) on average but degrade in rare worst cases, which almost never matters for interviews but is worth knowing.

STEP BY STEP

Watch Two Sum run on nums = [2, 7, 11, 15] with target = 9. We store each number as we pass it, so the check is a single lookup.

i	nums[i]	need = target - nums[i]	seen so far	action
0	2	7	{}	7 not seen, store 2 to 0
1	7	2	{2: 0}	2 is in seen, return [0, 1]

04

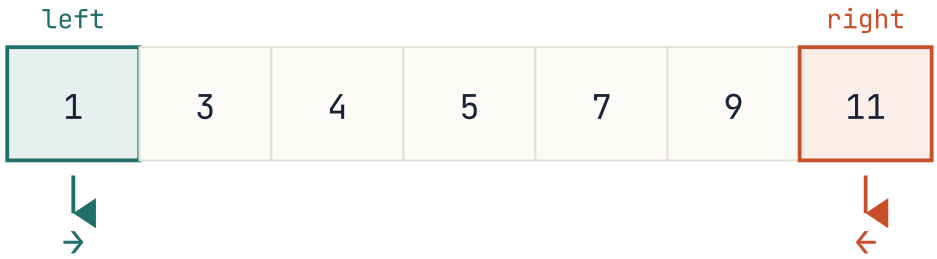
Two pointers

Instead of one index walking through an array, you use two, and you move them cleverly based on what you find. This turns many $O(n^2)$ brute forces into O(n), and it barely uses extra memory.

The two flavors

Opposite ends. One pointer starts at the left, one at the right, and they move toward each other. This is the go-to for sorted arrays and for anything symmetric like checking a palindrome.

Same direction (fast and slow). Both start at the left. One moves ahead to explore, the other lags behind to mark a boundary. This is how you remove duplicates in place or detect a cycle in a linked list.



sum too small? move left up. sum too big? move right down.

On a sorted array, comparing the current sum to the target tells you exactly which pointer to move.

REACH FOR TWO POINTERS WHEN YOU SEE

A sorted array plus "find a pair that sums to X". A palindrome check. "Remove duplicates in place." "Container with most water." "Move zeros to the end." Anything where you can make a decision by comparing the two ends, or where one runner explores while another marks position.

Worked example: two sum on a sorted array

Because the array is sorted, you do not need a hash map at all. Point at both ends. If the current sum is too small, the only way to grow it is to move the left pointer right (to a bigger number). If it is too big, move the right pointer left (to a smaller number). You can never miss the answer.

PYTHON

```python
def two_sum_sorted(nums, target):
    left, right = 0, len(nums) - 1
    while left < right:
        total = nums[left] + nums[right]
        if total == target:
            return [left, right]
        elif total < target:
            left += 1          # need a bigger sum
        else:
            right -= 1         # need a smaller sum
    return []
```

JAVA

```java
int[] twoSumSorted(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    while (left < right) {
        int total = nums[left] + nums[right];
        if (total == target) return new int[]{left, right};
        else if (total < target) left++;   // need a bigger sum
        else right--;                        // need a smaller sum
    }
    return new int[]{};
}
```

This is O(n) time and O(1) space. No extra map, just two indices.

Worked example: valid palindrome

Check whether a string reads the same forwards and backwards, ignoring case and non-letters. Point at both ends, skip anything that is not a letter or digit, and compare. Walk inward until the pointers meet.

PYTHON

```python
def is_palindrome(s):
    left, right = 0, len(s) - 1
    while left < right:
        while left < right and not s[left].isalnum():
            left += 1
        while left < right and not s[right].isalnum():
            right -= 1
        if s[left].lower() != s[right].lower():
            return False
        left += 1
        right -= 1
    return True
```

JAVA

```java
boolean isPalindrome(String s) {
    int left = 0, right = s.length() - 1;
    while (left < right) {
        while (left < right && !Character.isLetterOrDigit(s.charAt(left)))
left++;
        while (left < right && !Character.isLetterOrDigit(s.charAt(right)))
right--;
        if (Character.toLowerCase(s.charAt(left)) !=
            Character.toLowerCase(s.charAt(right))) return false;
        left++;
        right--;
    }
    return true;
}
```

The same-direction variant: remove duplicates in place

Given a sorted array, keep only one of each value, without allocating a new array. A slow pointer marks where the next unique value should go. A fast pointer scans ahead. Whenever fast finds something new, write it to the slow position and step slow forward.

PYTHON

```python
def remove_duplicates(nums):
    if not nums:
        return 0
    slow = 0                        # last position of a kept value
    for fast in range(1, len(nums)):
        if nums[fast] ≠ nums[slow]:
            slow += 1
            nums[slow] = nums[fast]
    return slow + 1                 # length of the unique part
```

JAVA

```java
int removeDuplicates(int[] nums) {
    if (nums.length == 0) return 0;
    int slow = 0;                           // last kept position
    for (int fast = 1; fast < nums.length; fast++) {
        if (nums[fast] ≠ nums[slow]) {
            slow++;
            nums[slow] = nums[fast];
        }
    }
    return slow + 1;                         // length of unique part
}
```

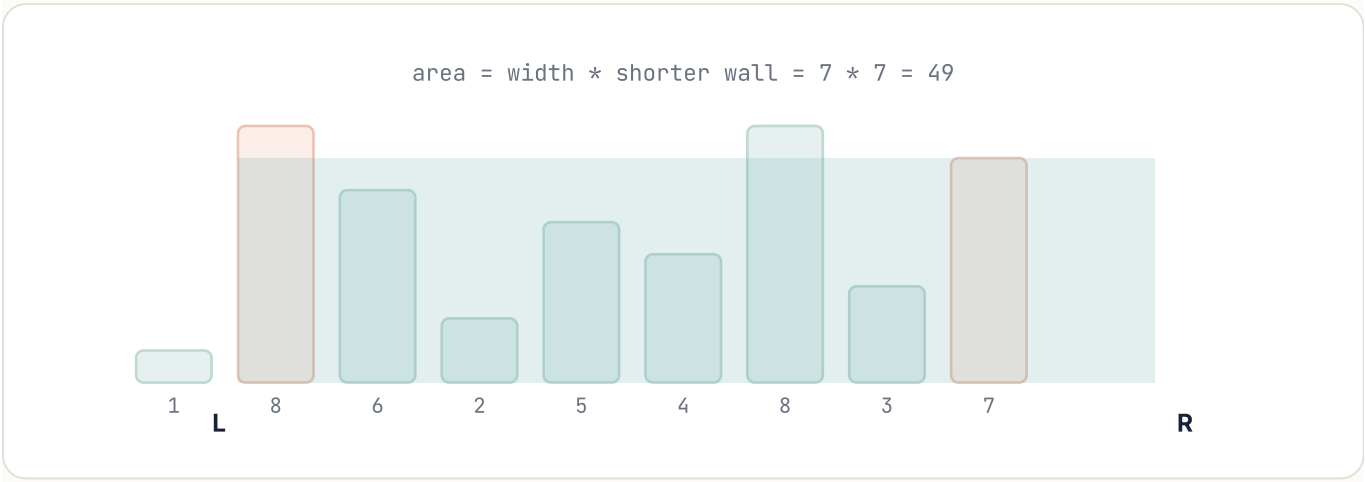
WATCH OUT

Opposite-ends two pointers usually requires the array to be sorted first. If it is not sorted and the problem does not let you sort, a hash map is often the better tool. Also be careful with the loop condition: `while left < right` stops the pointers from crossing, which is almost always what you want.

DEEPER INTUITION

Why moving the shorter side is safe

On a sorted array, the two ends carry information: if their sum is too big the answer must be smaller, so you can only help by pulling the right pointer in, and if it is too small you pull the left in. Each move permanently discards a whole set of pairs you never have to check again, which is why the work drops from O(n squared) to O(n). In the water container below, the height is limited by the shorter wall, so widening toward the taller wall can only lose area, and moving the shorter wall is the only move that can improve things.



Water is capped by the shorter wall, so moving the taller wall never helps. Move the shorter one inward.

Another worked example: Container With Most Water

Two walls hold water up to the height of the shorter one, times the distance between them. Start wide at both ends and always move the shorter wall inward, tracking the best area seen. Moving the shorter wall is the only choice that can ever increase the area.

PYTHON

```python
def max_area(height):
    left, right = 0, len(height) - 1
    best = 0
    while left < right:
        width = right - left
        best = max(best, width * min(height[left], height[right]))
        if height[left] < height[right]:    # move the shorter wall in
            left += 1
        else:
            right -= 1
    return best
```

JAVA

```java
int maxArea(int[] height) {
    int left = 0, right = height.length - 1, best = 0;
    while (left < right) {
        int width = right - left;
        best = Math.max(best, width * Math.min(height[left], height[right]));
        if (height[left] < height[right]) left++;   // move shorter wall
        else right--;
    }
    return best;
}
```

GOING DEEPER

What kinds of problems this solves

USE THIS PATTERN FOR

Problems on a sorted array or a string where two positions can move toward each other or in the same direction, so you replace a nested loop with a single sweep. Think pairing, partitioning in place, palindrome checks, and merging.

Problem type	Classic examples
Ends moving inward	Two Sum II, 3Sum, 4Sum, Container With Most Water, Trapping Rain Water, Valid Palindrome
Same direction, slow and fast	Remove Duplicates from Sorted Array, Move Zeroes, Remove Element, Sort Colors
Merging two sequences	Merge Sorted Array, Squares of a Sorted Array, Backspace String Compare

The algorithm, in a bit more detail

When the array is sorted, the two ends carry information. If the sum of the two ends is too big, the only way to get smaller is to pull the right pointer left. If it is too small, push the left pointer right. Each step throws away a whole set of pairs you no longer need to check, which is why the work drops from O(n squared) to O(n).

The same direction version uses a slow pointer that marks where the next good element goes and a fast pointer that scans ahead. This is how you compact an array in place, moving the elements you want to keep to the front without extra memory.

Variations you will run into

Common flavors are **opposite ends** for pair and area problems, **slow and fast in one direction** for in place filtering, and **three pointers** for 3Sum, where you fix one element and two pointer the rest. The Dutch national flag idea (Sort Colors) uses three pointers to partition into low, middle, and high in one pass.

Edge cases and gotchas

- The opposite ends trick needs a sorted array, so sort first if the input is not sorted and the problem allows it.
- In 3Sum and 4Sum, skip duplicate values on both pointers or you will emit the same triple many times.
- Watch the loop condition, usually left is strictly less than right, so you do not compare an element with itself.

- For in place removal, return the new length and remember the array tail past that length is garbage.

STEP BY STEP

Two pointers on the sorted array [1, 3, 4, 6, 8, 10] with target = 10. The ends move inward based on whether the sum is too big or too small.

left	right	sum	compare to 10	action
0	5	1 + 10 = 11	too big	move right in
0	4	1 + 8 = 9	too small	move left in
1	4	3 + 8 = 11	too big	move right in
1	3	3 + 6 = 9	too small	move left in
2	3	4 + 6 = 10	equal	found, return [2, 3]

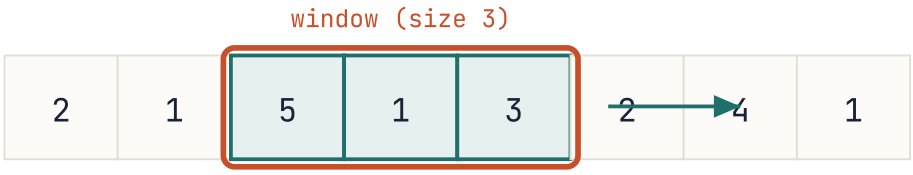
05 Sliding window

A window is a range of the array or string that you keep looking at. Instead of recomputing everything for every possible range, you slide the window along and only update what changed at the edges. This is how "check every substring" goes from $O(n^2)$ down to $O(n)$.

Two kinds of window

Fixed size. The window is always k wide. You slide it one step at a time: add the new element on the right, drop the old one on the left. Great for "max sum of any k consecutive elements."

Variable size. The window grows and shrinks based on a condition. You expand the right edge to include more, and when a rule is broken you shrink from the left until it holds again. Great for "longest substring with no repeats" or "smallest subarray with sum at least X."



sum here = 9

slide right: add the new number, subtract the one that left

When the window slides, only the two edge values change, so you never recount the middle.

REACH FOR A SLIDING WINDOW WHEN YOU SEE

"Longest" or "shortest" or "maximum" or "minimum" attached to a contiguous subarray or substring. Also anything with "at most K" or "exactly K" of something in a row. The key word is contiguous: the elements must be next to each other. If they can be scattered, this is not your pattern.

Fixed window: maximum sum of k consecutive elements

Compute the sum of the first k elements once. Then slide: each step, add the element entering the window and subtract the one leaving. One pass, O(n).

PYTHON

```python
def max_sum_k(nums, k):
    window = sum(nums[:k])         # first window
    best = window
    for i in range(k, len(nums)):
        window += nums[i] - nums[i - k]   # add new, drop old
        best = max(best, window)
    return best
```

JAVA

```java
int maxSumK(int[] nums, int k) {
    int window = 0;
    for (int i = 0; i < k; i++) window += nums[i];   // first window
    int best = window;
    for (int i = k; i < nums.length; i++) {
        window += nums[i] - nums[i - k];             // add new, drop old
        best = Math.max(best, window);
    }
    return best;
}
```

Variable window: longest substring without repeating characters

Grow the window to the right, one character at a time. Keep a set of what is currently inside. The moment the new character is already in the set, shrink from the left, removing characters until the duplicate is gone. Track the biggest window you ever had.

PYTHON

```python
def longest_unique(s):
    seen = set()
    left = 0
    best = 0
    for right in range(len(s)):
        while s[right] in seen:        # duplicate found, shrink left
            seen.remove(s[left])
            left += 1
        seen.add(s[right])             # now safe to add
        best = max(best, right - left + 1)
    return best
```

JAVA

```java
int longestUnique(String s) {
    Set<Character> seen = new HashSet<>();
    int left = 0, best = 0;
    for (int right = 0; right < s.length(); right++) {
        while (seen.contains(s.charAt(right))) {  // duplicate, shrink
            seen.remove(s.charAt(left));
            left++;
        }
        seen.add(s.charAt(right));
        best = Math.max(best, right - left + 1);
    }
    return best;
}
```

THE WINDOW TEMPLATE

Almost every variable window looks the same: a `for` loop moves `right` forward to expand, and an inner `while` moves `left` forward to shrink whenever the window becomes invalid. Update your answer either when the window is valid (for "longest") or right after fixing it (for "shortest"). Learn this shape once and dozens of problems fall out of it.

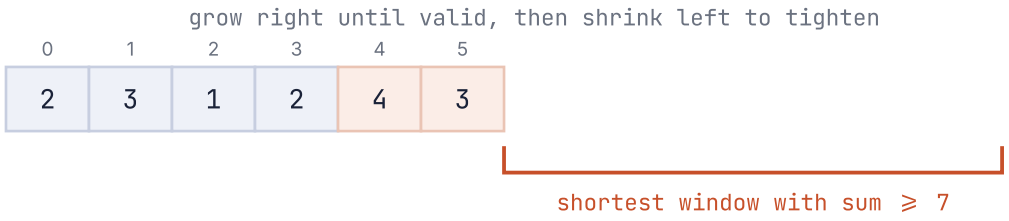
WATCH OUT

The window size is `right - left + 1`, not `right - left`. Off-by-one here is the most common bug. Also, for "shortest" problems you update the answer inside the shrink loop; for "longest" you update after the window is made valid again.

DEEPER INTUITION

Why it stays linear

The window is defined by a left and a right index, and a small running summary of what is inside. The right index only ever moves forward, and the left index only ever moves forward, so across the whole run each element is added once and removed at most once. That is why a problem that looks like it needs a nested scan finishes in a single O(n) pass. The only real decision is when to grow versus when to shrink.



For a shortest window, grow until the condition holds, then shrink from the left as far as you can.

Another worked example: Minimum Size Subarray Sum

Find the shortest contiguous subarray whose sum is at least a target. Grow the window by moving right and adding to a running total, and whenever the total is large enough, shrink from the left to make the window as short as possible while still valid, recording the best length.

PYTHON

```python
def min_subarray_len(target, nums):
    left = 0
    total = 0
    best = float('inf')
    for right, x in enumerate(nums):
        total += x                      # grow the window
        while total ⩾ target:           # shrink while still valid
            best = min(best, right - left + 1)
            total -= nums[left]
            left += 1
    return 0 if best == float('inf') else best
```

JAVA

```java
int minSubArrayLen(int target, int[] nums) {
    int left = 0, total = 0, best = Integer.MAX_VALUE;
    for (int right = 0; right < nums.length; right++) {
        total += nums[right];               // grow
        while (total ⩾ target) {            // shrink
            best = Math.min(best, right - left + 1);
            total -= nums[left++];
        }
    }
    return best == Integer.MAX_VALUE ? 0 : best;
}
```

GOING DEEPER

What kinds of problems this solves

USE THIS PATTERN FOR

Any question about the best or valid contiguous run in an array or string. Longest substring with some property, shortest subarray that reaches a sum, or a fixed size window you slide across and aggregate.

Problem type	Classic examples
Fixed size window	Maximum Average Subarray, Find All Anagrams in a String, Permutation in String
Longest valid window	Longest Substring Without Repeating Characters, Longest Repeating Character Replacement, Max Consecutive Ones III, Fruit Into Baskets
Shortest or at least window	Minimum Size Subarray Sum, Minimum Window Substring, Subarrays with K Different Integers

The algorithm, in a bit more detail

You keep a window defined by a left and a right index and a small summary of what is inside, often a running sum or a map of character counts. You always grow the window by moving right forward. The moment the window breaks the rule (a repeat appears, the sum gets too big, too many distinct letters), you shrink from the left until it is valid again. You update your answer as you go.

The reason this is O(n) and not O(n squared) is that each index is added when right passes it and removed at most once when left passes it. Every element enters and leaves the window a single time, so the total work is linear even though the window is constantly resizing.

Variations you will run into

Fixed windows are the easiest: slide a window of size k and keep a rolling sum. Variable windows split into "find the longest valid" (grow greedily, shrink only when broken) and "find the shortest that satisfies" (grow until satisfied, then shrink to tighten). A neat trick for "exactly K distinct" is to compute "at most K" minus "at most K minus 1".

Edge cases and gotchas

- Decide clearly when to shrink and whether you update the answer while the window is valid or while fixing it, since longest and shortest differ here.
- For substring problems, a count map plus a counter of how many characters still need matching is cleaner than rescanning.
- Do not forget to remove the character at the old left index from your summary when you shrink.

- Empty input and windows larger than the array are the usual off by one traps.

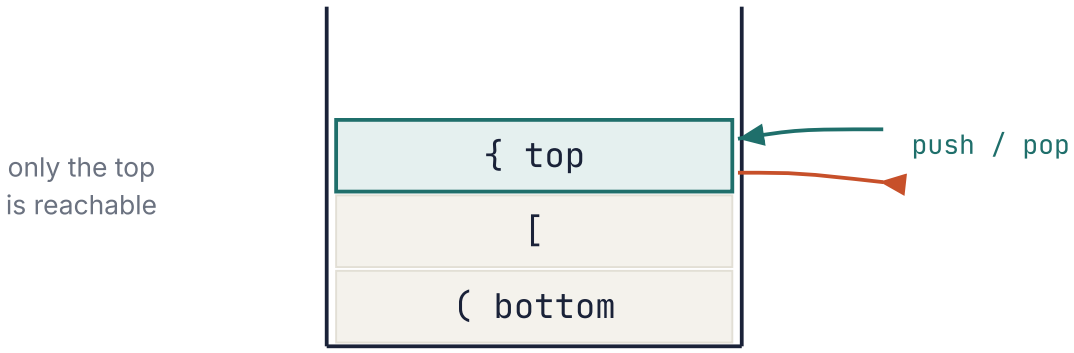
STEP BY STEP

The longest substring without repeats on 'abcabcbb'. The right edge grows the window, and a repeat forces the left edge to shrink until the window is valid again.

right	char	window	best length
0	a	a	1
1	b	ab	2
2	c	abc	3
3	a	abc → shrink → bca	3
4	b	cab	3
5	c	abc	3

06 Stack

A stack is a pile. You add to the top and you remove from the top, so the last thing you put in is the first thing you take out. That "last in, first out" rule turns out to be exactly what you need for matching brackets, undo history, and figuring out the "next bigger thing" in an array.



Push adds on top, pop removes from top. You never touch the middle.

REACH FOR A STACK WHEN YOU SEE

Matching or balancing brackets. "Evaluate this expression." "Undo the last action." Anything where you process items and sometimes need to cancel or pair with the most recent one. And a special kind called a monotonic stack for "next greater element" or "next smaller element" style problems.

Worked example: valid parentheses

Given a string of brackets, decide if they are correctly matched and nested. Every time you see an opening bracket, push it. Every time you see a closing bracket, the top of the stack must be its matching opener. If it is, pop it off. If not, or if the stack is empty when you need it, the string is invalid. At the end the stack must be empty.

PYTHON

```python
def is_valid(s):
    pairs = {')': '(', ']': '[', '}': '{'}
    stack = []
    for c in s:
        if c in pairs:                      # a closing bracket
            if not stack or stack[-1] ≠ pairs[c]:
                return False                # nothing to match, or wrong
match
            stack.pop()
        else:                               # an opening bracket
            stack.append(c)
    return not stack                        # valid only if nothing is left
```

JAVA

```java
boolean isValid(String s) {
    Map<Character, Character> pairs = Map.of(')', '(', ']', '[', '}', '{');
    Deque<Character> stack = new ArrayDeque<>();
    for (char c : s.toCharArray()) {
        if (pairs.containsKey(c)) {                 // closing bracket
            if (stack.isEmpty() || stack.pop() ≠ pairs.get(c)) return false;
        } else {
            stack.push(c);                          // opening bracket
        }
    }
    return stack.isEmpty();                          // must be empty
}
```

USE THE RIGHT STACK TYPE IN JAVA

Do not use the old `java.util.Stack` class; it is slower and synchronized. Use `ArrayDeque` instead, with `push`, `pop`, and `peek`. In Python, a plain list is already a perfect stack: `append` to push and `pop()` to pop.

The monotonic stack: next greater element

This is the stack trick that feels like magic. For each element, you want the next element to its right that is bigger than it. Brute force is $O(n^2)$. With a stack that stays in decreasing order, you get $O(n)$.

The idea: walk left to right, keeping a stack of indices whose answer you have not found yet. When the current number is bigger than the value at the top of the stack, it is the answer for that top element, so pop it and record it. Keep popping while the current number is bigger. Then push the current index.

PYTHON

```python
def next_greater(nums):
    result = [-1] * len(nums)     # default: no greater element
    stack = []                    # indices waiting for their answer
    for i, x in enumerate(nums):
        while stack and nums[stack[-1]] < x:
            result[stack.pop()] = x   # x is the next greater for that index
        stack.append(i)
    return result
```

JAVA

```java
int[] nextGreater(int[] nums) {
    int n = nums.length;
    int[] result = new int[n];
    Arrays.fill(result, -1);              // default: none
    Deque<Integer> stack = new ArrayDeque<>();  // indices waiting
    for (int i = 0; i < n; i++) {
        while (!stack.isEmpty() && nums[stack.peek()] < nums[i]) {
            result[stack.pop()] = nums[i];
        }
        stack.push(i);
    }
    return result;
}
```

Each index is pushed once and popped at most once, so even though there is a loop inside a loop, the total work is $O(n)$. That is the payoff of the monotonic stack: it looks quadratic but it is linear.

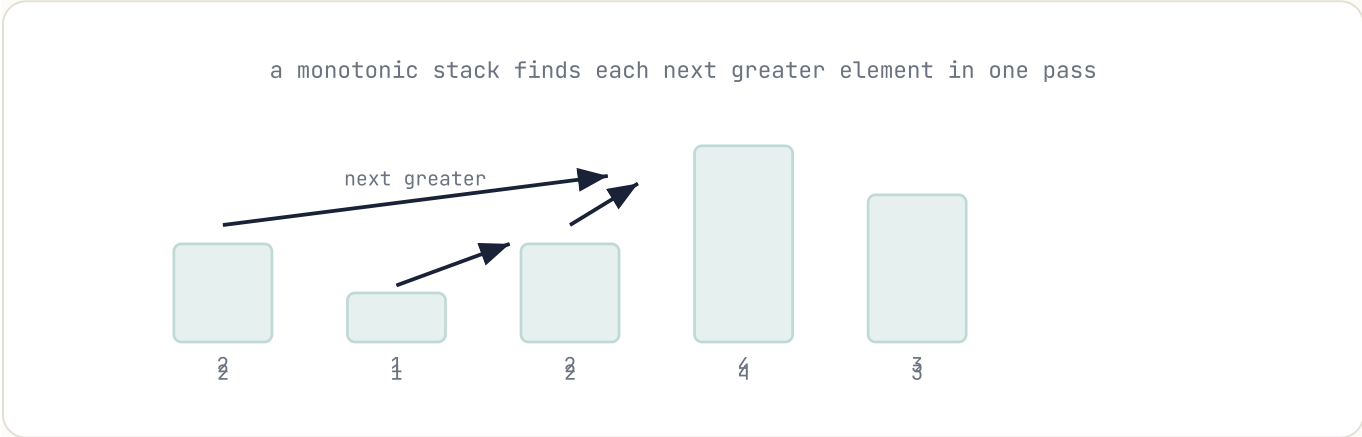
WATCH OUT

Decide up front whether your stack holds values or indices. For "next greater" you almost always want indices, so you can write the answer back into the correct slot. Storing values loses the position and forces you to redo work.

DEEPER INTUITION

Why the monotonic stack is O(n)

A monotonic stack keeps its contents in order, here strictly decreasing. When a new element arrives that is bigger than the top, it pops every smaller element, and each pop is the exact moment that popped element has found its next greater value. Because every index is pushed once and popped at most once, the whole scan is linear even though it feels like it looks backward for each element.



Each element waits on the stack until a bigger one arrives, which is the moment its answer is found.

Another worked example: Daily Temperatures

For each day, how many days until a warmer one. Keep a stack of day indices whose temperatures are decreasing. When today is warmer than the day on top, pop it and record the gap, since today is that day's answer. Whatever never gets popped simply has no warmer day ahead.

PYTHON

```python
def daily_temperatures(temps):
    answer = [0] * len(temps)
    stack = []                          # indices, temperatures decreasing
    for i, t in enumerate(temps):
        while stack and temps[stack[-1]] < t:
            j = stack.pop()             # today is warmer than day j
            answer[j] = i - j           # how long day j waited
        stack.append(i)
    return answer
```

JAVA

```java
int[] dailyTemperatures(int[] temps) {
    int[] answer = new int[temps.length];
    Deque<Integer> stack = new ArrayDeque<>();   // decreasing temps
    for (int i = 0; i < temps.length; i++) {
        while (!stack.isEmpty() && temps[stack.peek()] < temps[i]) {
            int j = stack.pop();
            answer[j] = i - j;
        }
        stack.push(i);
    }
    return answer;
}
```

GOING DEEPER

What kinds of problems this solves

USE THIS PATTERN FOR

Anything with nesting or matching, anything that needs the most recent unfinished thing first, and the whole family of next greater or next smaller questions solved with a monotonic stack.

Problem type	Classic examples
Matching and nesting	Valid Parentheses, Minimum Remove to Make Valid, Simplify Path, Basic Calculator
Monotonic stack	Next Greater Element, Daily Temperatures, Largest Rectangle in Histogram, Trapping Rain Water, Online Stock Span
Stack as history or undo	Backspace String Compare, Decode String, Asteroid Collision, Remove All Adjacent Duplicates

The algorithm, in a bit more detail

A stack gives you last in first out access, which is exactly what nesting needs. Every open bracket waits on the stack until its matching close arrives. If a close does not match the top, the string is invalid. This same "wait until resolved" idea powers path simplification and expression evaluation.

A monotonic stack keeps its contents sorted, either always increasing or always decreasing. When a new element would break that order, you pop everything it beats, and each pop is the moment you have found that popped element's answer, like its next greater value or the boundary of a rectangle. Every element is pushed once and popped once, so the whole scan is O(n) even though it feels like it is looking backward.

Variations you will run into

The two big variants are the **plain stack** for matching and history, and the **monotonic stack** for range and next greater problems. In monotonic problems you often store indices rather than values, so you can compute distances and widths when you pop.

Edge cases and gotchas

- Always check the stack is not empty before you peek or pop.
- For monotonic problems, decide up front whether you want increasing or decreasing order, and whether you push indices or values.
- In Java prefer Deque over the legacy Stack class, since Stack is synchronized and slower.

- Remember to handle whatever is left on the stack after the loop, since leftovers often mean unmatched items.

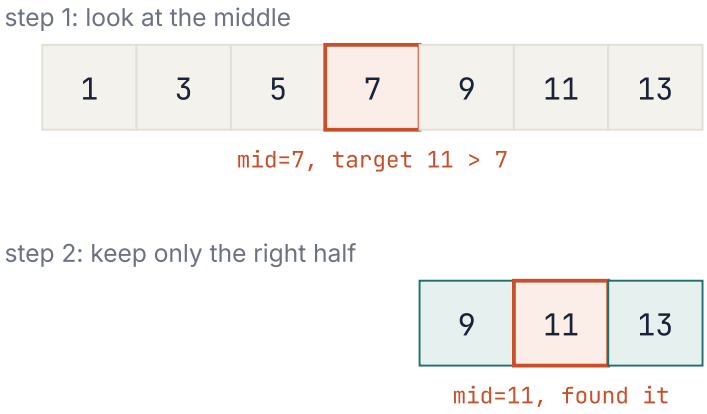
STEP BY STEP

Valid parentheses on the string '([])'. Each opener is pushed and waits, and each closer must match the top of the stack.

char	action	stack after
(	push	(
[	push	([
]	top is [, match, pop	(
)	top is (, match, pop	empty
end	stack empty, so valid	empty

07 Binary search

If your data is sorted, you never need to check every element. Look at the middle. If it is too small, throw away the left half. If it is too big, throw away the right half. Each step halves what is left, so even a billion items take only about thirty checks.



Each comparison discards half the remaining elements. That is why it is O(log n).

REACH FOR BINARY SEARCH WHEN YOU SEE

A sorted array plus "find X" or "find the position where X would go." Also the sneaky version: "find the smallest value that satisfies some condition" or "minimize the maximum." If you can phrase the problem as "is a candidate answer good enough, yes or no," and the yes/no flips at one point, you can binary search on the answer itself even when there is no array to search.

The classic template

Keep two bounds, `lo` and `hi`. Look at the middle. Move whichever bound is wrong. The one detail people get wrong is computing the middle safely and updating the bounds so the loop always shrinks.

PYTHON

```python
def binary_search(nums, target):
    lo, hi = 0, len(nums) - 1
    while lo ⩽ hi:
        mid = (lo + hi) // 2         # middle index
        if nums[mid] == target:
            return mid
        elif nums[mid] < target:
            lo = mid + 1             # answer is to the right
        else:
            hi = mid - 1             # answer is to the left
    return -1                        # not found
```

JAVA

```java
int binarySearch(int[] nums, int target) {
    int lo = 0, hi = nums.length - 1;
    while (lo ⩽ hi) {
        int mid = lo + (hi - lo) / 2;    // avoids integer overflow
        if (nums[mid] == target) return mid;
        else if (nums[mid] < target) lo = mid + 1;  // go right
        else hi = mid - 1;                           // go left
    }
    return -1;
}
```

COMPUTE MID WITHOUT OVERFLOW

In Java, `(lo + hi) / 2` can overflow when both are large, because the sum exceeds the max `int`. Write `lo + (hi - lo) / 2` instead. Python integers never overflow, so either form is fine there, but the habit is good.

Binary search on the answer

This is the version that separates people who "know binary search" from people who really get it. Example: you must ship packages within D days, and you choose a daily capacity. Bigger capacity finishes faster. You want the smallest capacity that still finishes within D days.

There is no array to search. But the space of possible capacities is a sorted range, and "can we finish in D days with capacity C?" is a yes/no question that becomes yes once C is large enough. So binary search on C.

PYTHON

```python
def ship_within_days(weights, days):
    def can_do(cap):
        needed, load = 1, 0
        for w in weights:
            if load + w > cap:      # start a new day
                needed += 1
                load = 0
            load += w
        return needed ⩽ days

    lo, hi = max(weights), sum(weights)   # smallest and largest sane
capacity
    while lo < hi:
        mid = (lo + hi) // 2
        if can_do(mid):
            hi = mid                # mid works, try smaller
        else:
            lo = mid + 1            # mid too small, go bigger
    return lo
```

JAVA

```java
int shipWithinDays(int[] weights, int days) {
    int lo = 0, hi = 0;
    for (int w : weights) { lo = Math.max(lo, w); hi += w; }
    while (lo < hi) {
        int mid = lo + (hi - lo) / 2;
        if (canDo(weights, days, mid)) hi = mid;   // works, try smaller
        else lo = mid + 1;                          // too small, go bigger
    }
    return lo;
}
boolean canDo(int[] weights, int days, int cap) {
    int needed = 1, load = 0;
    for (int w : weights) {
        if (load + w > cap) { needed++; load = 0; }
        load += w;
    }
    return needed ⩽ days;
}
```

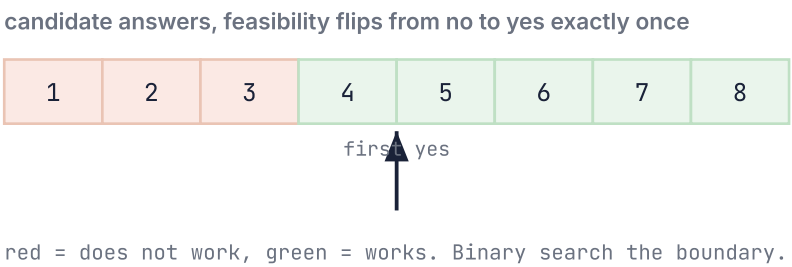
WATCH OUT

Mixing up the loop condition and the update is the classic infinite loop. If you use `while lo < hi` with `hi = mid`, you must pair it with `lo = mid + 1`, or the range may never shrink. When in doubt, trace a two-element case by hand to be sure the window always gets smaller.

DEEPER INTUITION

Why you can search the answer, not just the array

Binary search needs one thing: a way to look at the middle and confidently throw away half. In a sorted array a comparison gives you that. But the same trick works on the space of possible answers whenever a yes or no test is monotonic, meaning once some value works, every larger value also works. Then you binary search for the smallest value that works. Spotting that hidden monotonic property is what unlocks a whole class of hard problems.



When a yes or no test flips once as the value grows, you can binary search for the boundary.

Another worked example: Koko Eating Bananas

Koko picks an eating speed and must finish all piles within a number of hours. A faster speed always finishes in the same or fewer hours, so feasibility is monotonic. Binary search the smallest speed for which the hours needed still fits the limit.

PYTHON

```python
def min_eating_speed(piles, hours):
    def hours_needed(speed):
        return sum((p + speed - 1) // speed for p in piles)   # ceil divide

    lo, hi = 1, max(piles)
    while lo < hi:
        mid = (lo + hi) // 2
        if hours_needed(mid) ⩽ hours:     # mid works, try slower
            hi = mid
        else:
            lo = mid + 1                  # too slow, go faster
    return lo
```

JAVA

```java
int minEatingSpeed(int[] piles, int hours) {
    int lo = 1, hi = 0;
    for (int p : piles) hi = Math.max(hi, p);
    while (lo < hi) {
        int mid = lo + (hi - lo) / 2;
        long need = 0;
        for (int p : piles) need += (p + mid - 1) / mid;   // ceil
        if (need ⩽ hours) hi = mid;       // works, try slower
        else lo = mid + 1;                // too slow
    }
    return lo;
}
```

GOING DEEPER

What kinds of problems this solves

USE THIS PATTERN FOR

Searching sorted data, finding the first or last position that satisfies a condition, and the powerful trick of searching over an answer space when the answer has a yes or no property that flips once as the value grows.

Problem type	Classic examples
Classic and boundaries	Binary Search, Search Insert Position, Find First and Last Position, Find Peak Element
Rotated and 2D	Search in Rotated Sorted Array, Find Minimum in Rotated Sorted Array, Search a 2D Matrix
Search on the answer	Koko Eating Bananas, Capacity To Ship Packages Within D Days, Split Array Largest Sum, Minimum Number of Days to Make Bouquets

The algorithm, in a bit more detail

Binary search keeps a range that is guaranteed to contain the answer and halves it every step. The trick is the invariant: you must be able to look at the middle and confidently throw away one half. In a sorted array that is easy, since a comparison tells you which side the target is on.

Search on the answer is the version that unlocks hard problems. Instead of searching the array, you search the range of possible answers, say the smallest eating speed or the smallest ship capacity. You write a helper that asks "does answer x work", which must be monotonic: if x works, everything bigger works too. Then you binary search for the smallest x that works. Recognizing that hidden monotonic property is the whole skill.

Variations you will run into

You will meet **leftmost and rightmost** variants (lower bound and upper bound), where a match does not stop the search but nudges a boundary to keep looking. Rotated arrays add one comparison to

decide which half is sorted. The answer space version replaces the array comparison with a feasibility check.

Edge cases and gotchas

- Compute the middle as low plus half the gap, not low plus high divided by two, to avoid integer overflow in Java.
- Match your loop condition and your boundary updates, since a wrong pairing causes an infinite loop or a missed answer.
- For first and last position, do not return on the first match, keep shrinking toward the side you want.
- Double check whether your interval is inclusive on both ends, since that decides less than versus less than or equal.

STEP BY STEP

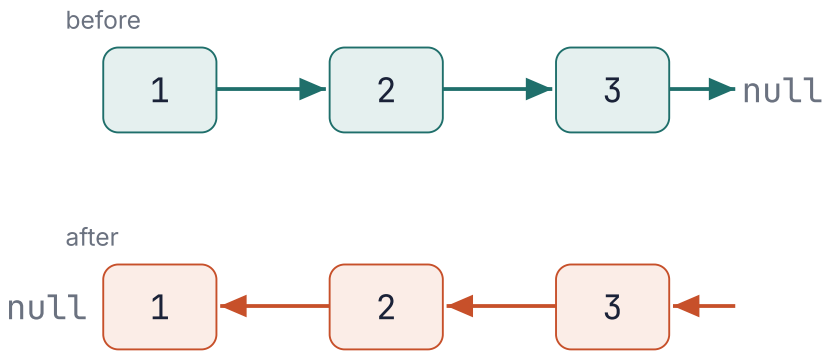
Searching for 7 in the sorted array [1, 3, 5, 7, 9, 11]. Each step throws away half of what is left.

low	high	mid	value at mid	action
0	5	2	5	5 is below 7, search right, low = 3
3	5	4	9	9 is above 7, search left, high = 3
3	3	3	7	7 equals target, return index 3

LINEAR AND LINKED

08 Linked lists

A linked list is a chain of nodes. Each node holds a value and a pointer to the next node. There is no index and no jumping around; to reach the fifth node you walk through the first four. The whole skill here is managing pointers without losing the chain.



Reversing means flipping every arrow to point backward, one node at a time.

REACH FOR LINKED LIST TECHNIQUES WHEN YOU SEE

"Reverse the list", "find the middle", "detect a cycle", "merge two sorted lists", "remove the Nth node from the end." The two workhorses are the fast-and-slow pointer (for middle and cycle) and the dummy head node (to simplify insertions and deletions at the front).

The node definition

PYTHON

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next
```

JAVA

```java
class ListNode {
    int val;
    ListNode next;
    ListNode(int val) { this.val = val; }
}
```

Worked example: reverse a linked list

Walk the list. At each node, remember the next node, then point the current node backward at the previous node, then step everything forward. The trap is losing the rest of the chain, so you must save `next` before you overwrite it.

PYTHON

```python
def reverse_list(head):
    prev = None
    curr = head
    while curr:
        nxt = curr.next      # save the rest of the chain first
        curr.next = prev     # flip the arrow backward
        prev = curr          # move prev forward
        curr = nxt           # move curr forward
    return prev              # prev is the new head
```

JAVA

```java
ListNode reverseList(ListNode head) {
    ListNode prev = null, curr = head;
    while (curr ≠ null) {
        ListNode next = curr.next;   // save the rest first
        curr.next = prev;            // flip the arrow
        prev = curr;                 // advance prev
        curr = next;                 // advance curr
    }
    return prev;                     // new head
}
```

Fast and slow pointers: find the middle, and detect a cycle

Move one pointer one step at a time and another two steps at a time. When the fast one reaches the end, the slow one is exactly in the middle. And if the list secretly loops back on itself, the fast pointer will eventually lap the slow one and they will meet. This is called Floyd's cycle detection, and it uses no extra memory.

PYTHON

```python
def find_middle(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next          # one step
        fast = fast.next.next     # two steps
    return slow                   # middle node

def has_cycle(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow is fast:          # they met, there is a loop
            return True
    return False
```

JAVA

```java
ListNode findMiddle(ListNode head) {
    ListNode slow = head, fast = head;
    while (fast ≠ null && fast.next ≠ null) {
        slow = slow.next;         // one step
        fast = fast.next.next;    // two steps
    }
    return slow;                  // middle
}

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;  // they met
    }
    return false;
}
```

The dummy node trick: merge two sorted lists

When you build a new list or delete from the front, having a fake node in front of the real head saves you from special-casing the first element. You always attach to `tail.next` and return `dummy.next` at the end.

PYTHON

```python
def merge_two_lists(a, b):
    dummy = ListNode()        # fake head so we never special-case the start
    tail = dummy
    while a and b:
        if a.val ⩽ b.val:
            tail.next = a
            a = a.next
        else:
            tail.next = b
            b = b.next
        tail = tail.next
    tail.next = a or b        # attach whatever is left
    return dummy.next
```

JAVA

```java
ListNode mergeTwoLists(ListNode a, ListNode b) {
    ListNode dummy = new ListNode(0);   // fake head
    ListNode 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;    // attach the remainder
    return dummy.next;
}
```

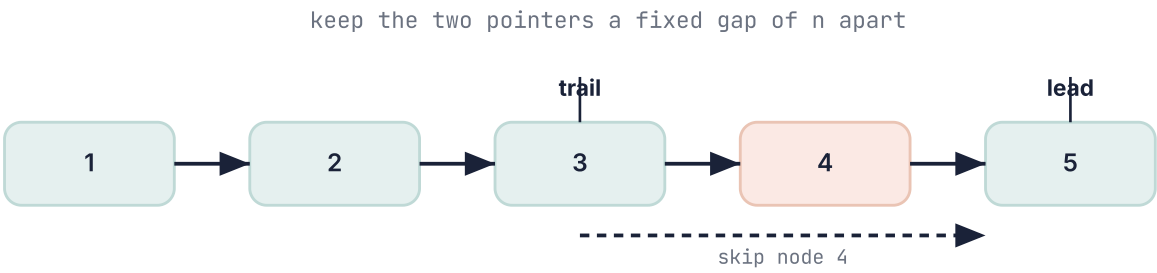
WATCH OUT

The number one linked list bug is a null pointer error. Before you write `node.next.next`, be sure `node.next` is not null. In the fast-and-slow loop, always check `fast and fast.next` before stepping, in that order. And when reordering pointers, save what you need before you overwrite it, or the tail of your list vanishes.

DEEPER INTUITION

Why a gap of n finds the end

Two pointers that stay exactly n nodes apart turn "nth from the end" into a single pass. Start the lead pointer n steps ahead, then move both together. When the lead reaches the last node, the trailing pointer is sitting right before the node you want to remove, because the fixed gap guarantees it. A dummy head in front of the list means even removing the first node needs no special case.



Start one pointer n nodes ahead. When it reaches the end, the other sits just before the target.

Another worked example: Remove Nth Node From End

Delete the nth node counting from the end in one pass. Put a lead pointer n steps ahead of a trailing pointer, then advance both until the lead hits the end. The trailing pointer now points at the node just before the target, so you skip it.

PYTHON

```python
def remove_nth_from_end(head, n):
    dummy = ListNode(0, head)
    lead = trail = dummy
    for _ in range(n):
        lead = lead.next            # move lead n steps ahead
    while lead.next:                # move both until lead is last
        lead = lead.next
        trail = trail.next
    trail.next = trail.next.next    # skip the target node
    return dummy.next
```

JAVA

```java
ListNode removeNthFromEnd(ListNode head, int n) {
    ListNode dummy = new ListNode(0, head);
    ListNode lead = dummy, trail = dummy;
    for (int i = 0; i < n; i++) lead = lead.next;   // n ahead
    while (lead.next ≠ null) {                      // move together
        lead = lead.next;
        trail = trail.next;
    }
    trail.next = trail.next.next;                   // skip target
    return dummy.next;
}
```

GOING DEEPER

What kinds of problems this solves

USE THIS PATTERN FOR

Problems where you rewire pointers rather than move data: reversing, detecting or locating a cycle, finding the middle or the kth from the end, and merging or reordering lists.

Problem type	Classic examples
Pointer rewiring	Reverse Linked List, Reverse Nodes in k-Group, Swap Nodes in Pairs, Rotate List
Fast and slow pointers	Middle of the Linked List, Linked List Cycle, Linked List Cycle II, Palindrome Linked List
Merging with a dummy node	Merge Two Sorted Lists, Merge k Sorted Lists, Add Two Numbers, Remove Nth Node From End of List

The algorithm, in a bit more detail

Two ideas cover most linked list problems. The first is the dummy head: you make a fake node that points at the real head, build your result off the dummy, and return dummy.next. This removes all the annoying special cases around an empty list or changing the first node.

The second is two pointers with a gap or a speed difference. A fast pointer that moves two steps and a slow pointer that moves one will meet inside any cycle, and the slow pointer lands on the middle when the fast one reaches the end. To find the kth node from the end, start one pointer k steps ahead, then move both together until the leader hits the end.

Variations you will run into

Reversal shows up either **iteratively** with three pointers (previous, current, next) or **recursively**. Cycle problems use **Floyd's tortoise and hare**, and finding the cycle start needs the second phase where you reset one pointer to the head. Merging always benefits from the dummy node.

Edge cases and gotchas

- Save the next pointer before you overwrite it during reversal, or you will lose the rest of the list.
- Guard against null at every step, especially the fast pointer moving two at a time.
- Use a dummy head whenever the first node might change, so you never special case the head.
- After building a new list, remember to terminate it by setting the final next to null when needed.

STEP BY STEP

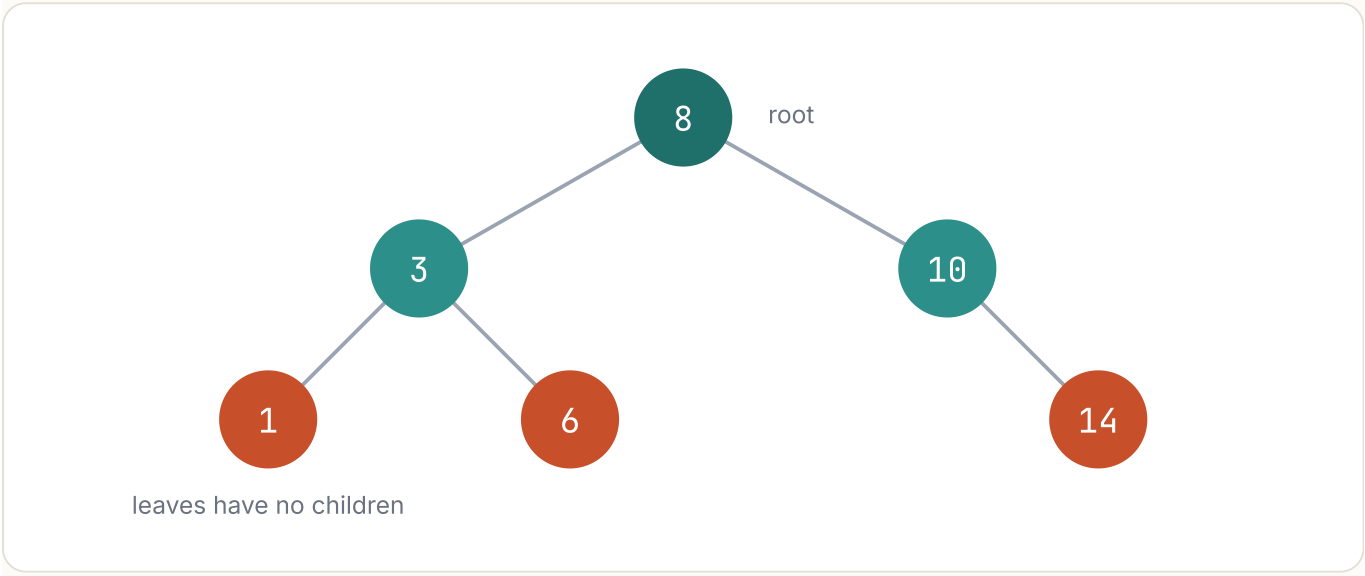
Reversing the list 1 to 2 to 3. We keep a previous pointer and, at each node, flip its next to point backward before moving on.

prev	current	action	list built so far
`null`	`1`	`1.next = null, prev = 1, current = 2`	`null ← 1`
`1`	`2`	`2.next = 1, prev = 2, current = 3`	`null ← 1 ← 2`
`2`	`3`	`3.next = 2, prev = 3, current = null`	`null ← 1 ← 2 ← 3`
`3`	`null`	`current is null, stop, return prev`	`new head is 3`

TREES AND HEAPS

09 Trees and BST

A tree is a branching structure. One node at the top (the root), and each node points to its children below. There are no loops. Most tree problems are solved by recursion, because a tree is made of smaller trees: every node is itself the root of its own subtree.



A binary tree: each node has at most a left child and a right child.

The two ways to walk a tree

Depth first (DFS). Go as deep as you can down one branch before backing up. This is naturally recursive. There are three orders depending on when you visit the node relative to its children:

- **Preorder:** node, then left, then right. Good for copying a tree.
- **Inorder:** left, then node, then right. On a binary search tree this visits values in sorted order.
- **Postorder:** left, then right, then node. Good when you need results from children first, like computing height.

Breadth first (BFS). Visit the tree level by level, top to bottom. This uses a queue, and it is the tool for "level order" problems and shortest path in an unweighted graph.

REACH FOR THESE WHEN YOU SEE

"Depth", "height", "count nodes", "is the tree balanced", "path sum": use DFS recursion. "Level order", "nodes at each level", "minimum depth", "right side view": use BFS with a queue. "Search / insert / validate" on a binary search tree: use the sorted property to go left or right.

The node, and DFS with recursion

Max depth is the perfect first example. The depth of a tree is one plus the depth of its taller subtree. An empty tree has depth zero. That sentence is the whole algorithm.

PYTHON

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def max_depth(root):
    if not root:
        return 0                        # empty tree has depth 0
    left = max_depth(root.left)         # depth of left subtree
    right = max_depth(root.right)       # depth of right subtree
    return 1 + max(left, right)         # this node adds one level
```

JAVA

```java
class TreeNode {
    int val;
    TreeNode left, right;
    TreeNode(int val) { this.val = val; }
}

int maxDepth(TreeNode root) {
    if (root == null) return 0;             // empty → 0
    int left = maxDepth(root.left);
    int right = maxDepth(root.right);
    return 1 + Math.max(left, right);       // this node adds a level
}
```

THE RECURSIVE TREE TEMPLATE

Almost every DFS tree function has the same three parts. First, the base case: if the node is null, return the "empty" answer (0, or true, or null). Second, recurse on the left and right children. Third, combine their answers with the current node's value. Once you see this shape, you can write most tree solutions in a minute.

BFS: level order traversal

Use a queue. Put the root in. Then, while the queue is not empty, process everything currently in it as one level, and add all their children for the next level. Counting the queue size at the start of each round is the trick that separates the levels cleanly.

PYTHON

```python
from collections import deque

def level_order(root):
    if not root:
        return []
    result = []
    queue = deque([root])
    while queue:
        level = []
        for _ in range(len(queue)):   # exactly the nodes on this level
            node = queue.popleft()
            level.append(node.val)
            if node.left:  queue.append(node.left)
            if node.right: queue.append(node.right)
        result.append(level)
    return result
```

JAVA

```java
List<List<Integer>> levelOrder(TreeNode root) {
    List<List<Integer>> result = new ArrayList<>();
    if (root == null) return result;
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    while (!queue.isEmpty()) {
        int size = queue.size();          // nodes on this level
        List<Integer> level = new ArrayList<>();
        for (int i = 0; i < size; i++) {
            TreeNode node = queue.poll();
            level.add(node.val);
            if (node.left != null)  queue.offer(node.left);
            if (node.right != null) queue.offer(node.right);
        }
        result.add(level);
    }
    return result;
}
```

Binary search trees: the sorted shortcut

A binary search tree keeps a rule at every node: everything in the left subtree is smaller, everything in the right subtree is bigger. That means searching is just repeated "go left or go right," which is binary search on a tree. Insert follows the same path until it finds an empty spot.

PYTHON

```python
def search_bst(root, target):
    while root:
        if target == root.val:
            return root
        root = root.left if target < root.val else root.right
    return None

def insert_bst(root, val):
    if not root:
        return TreeNode(val)          # found the empty spot
    if val < root.val:
        root.left = insert_bst(root.left, val)
    else:
        root.right = insert_bst(root.right, val)
    return root
```

JAVA

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

TreeNode insertBST(TreeNode root, int val) {
    if (root == null) return new TreeNode(val);  // empty spot
    if (val < root.val) root.left = insertBST(root.left, val);
    else root.right = insertBST(root.right, val);
    return root;
}
```

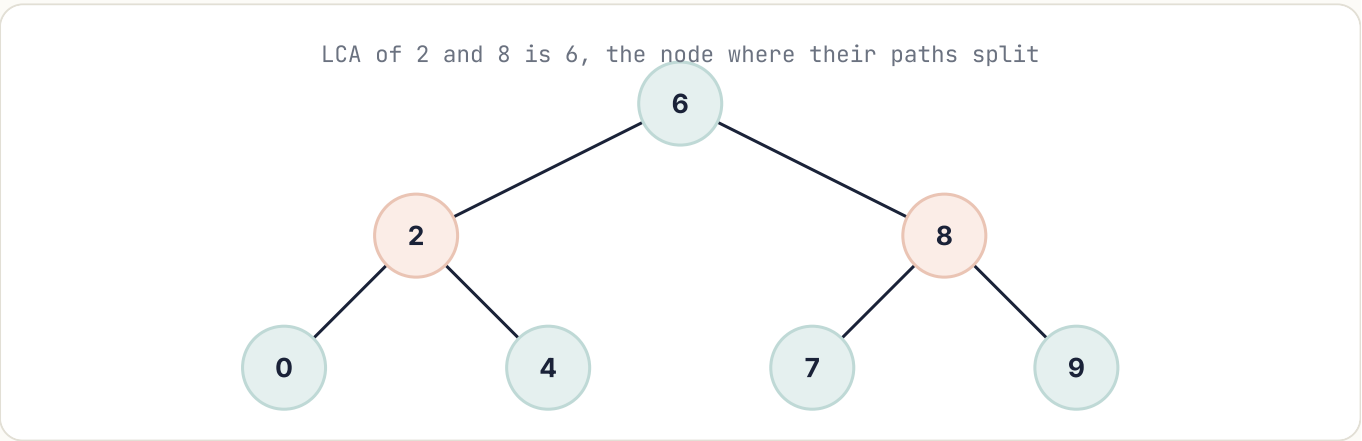
WATCH OUT

To validate a binary search tree, checking only that each node is bigger than its left child and smaller than its right child is not enough. A node deep on the left must be smaller than an ancestor far above it. Pass down a valid range (a low and high bound) as you recurse, and tighten it at each step. Also remember deep recursion can overflow the call stack on a very tall tree, so extremely skewed trees sometimes need an iterative approach.

DEEPER INTUITION

Why recursion and the BST order do the work

Most tree problems are solved by asking one question: if I already had the answers for my left and right subtrees, how would I combine them into mine. That single idea gives height, diameter, balance, and path sums almost for free. A binary search tree adds an ordering rule, everything left is smaller and everything right is larger, which lets you find, insert, or locate a common ancestor by simply comparing values and walking down one side.



In a BST, walk down comparing values. The first node that sits between the two targets is their lowest common ancestor.

Another worked example: Lowest Common Ancestor of a BST

The lowest common ancestor of two nodes is where their paths from the root first split. In a BST you can find it without recursion: walk down from the root, and if both targets are smaller go left, if both

are larger go right, otherwise this node is the split point and the answer.

PYTHON

```python
def lowest_common_ancestor(root, p, q):
    node = root
    while node:
        if p.val < node.val and q.val < node.val:
            node = node.left        # both smaller, go left
        elif p.val > node.val and q.val > node.val:
            node = node.right       # both larger, go right
        else:
            return node             # the split point is the LCA
```

JAVA

```java
TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
    TreeNode node = root;
    while (node ≠ null) {
        if (p.val < node.val && q.val < node.val) node = node.left;
        else if (p.val > node.val && q.val > node.val) node = node.right;
        else return node;           // split point
    }
    return null;
}
```

GOING DEEPER

What kinds of problems this solves

USE THIS PATTERN FOR

Anything on a hierarchy: depth and height, path and sum questions, validating or searching a binary search tree, working level by level, finding a lowest common ancestor, or building and serializing a tree.

Problem type	Classic examples
Depth and path with DFS	Maximum Depth, Diameter of Binary Tree, Path Sum, Balanced Binary Tree, Binary Tree Maximum Path Sum
Level by level with BFS	Binary Tree Level Order Traversal, Zigzag Level Order, Right Side View, Average of Levels
Binary search tree properties	Validate Binary Search Tree, Kth Smallest Element in a BST, Lowest Common Ancestor of a BST, Convert Sorted Array to BST
Construct and serialize	Construct Tree from Preorder and Inorder, Serialize and Deserialize Binary Tree, Flatten Binary Tree to Linked List

The algorithm, in a bit more detail

Most tree problems are a recursion that solves the children first and then combines their answers. Ask "if I already knew the answer for my left and right subtrees, how would I compute mine?" That single question gives you height, diameter, balance, and path sums almost for free.

A binary search tree adds an ordering rule: everything left is smaller, everything right is larger. That lets you search, insert, and find a lowest common ancestor by simply comparing values and walking down one side, which is O(height). An in order traversal of a BST visits values in sorted order, which is why "kth smallest" and "validate BST" lean on it.

Variations you will run into

Depth first traversal comes in **preorder, inorder, and postorder**, done recursively or with an explicit stack. Breadth first uses a **queue** and naturally groups nodes by level. For BST questions, always think about the ordering invariant before writing code.

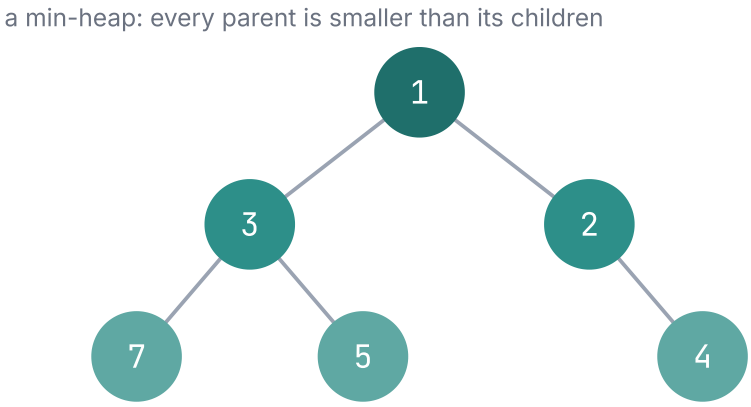
Edge cases and gotchas

- To validate a BST you must pass down an allowed range, not just compare a node with its direct parent.
- Handle the null node as your base case cleanly, it is the root of most tree bugs.

- Level order needs to capture the queue size at the start of each level so you know where one level ends.
- Recursion depth can be large on a skewed tree, so very deep trees may need an iterative version.

10 Heaps and priority queues

A heap is a container that always hands you the smallest (or largest) item first, and lets you add new items cheaply. It is the perfect tool whenever you repeatedly need "the best remaining thing" without sorting everything each time.



The smallest value is always at the top. Adding or removing takes O(log n).

REACH FOR A HEAP WHEN YOU SEE

"Kth largest" or "Kth smallest". "Top K frequent". "Merge K sorted lists". "Find the running median". Anything where you keep needing the current best or worst item as data streams in. If you catch yourself wanting to sort inside a loop, a heap is almost always the fix.

The one thing to remember about the two languages

Python's `heapq` is a min-heap: it gives you the smallest first. To get a max-heap, push the negatives of your numbers. Java's `PriorityQueue` is a min-heap by default too, but you can pass a comparator to flip it to a max-heap.

Worked example: Kth largest element

You want the Kth biggest number. The clean trick is to keep a min-heap of only the K largest numbers seen so far. Whenever it grows past K, pop the smallest. At the end, the smallest thing in that heap of size K is exactly the Kth largest overall. This runs in O(n log k), better than sorting the whole array when k is small.

PYTHON

```python
import heapq

def find_kth_largest(nums, k):
    heap = []                       # min-heap holding the k biggest so far
    for x in nums:
        heapq.heappush(heap, x)
        if len(heap) > k:
            heapq.heappop(heap)     # drop the smallest, keep the top k
    return heap[0]                  # smallest of the top k = kth largest
```

JAVA

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

Worked example: top K frequent elements

Count how often each value appears, then use a heap to pull out the K most frequent. This pairs the frequency-counting idea from the arrays chapter with a heap.

PYTHON

```python
import heapq
from collections import Counter

def top_k_frequent(nums, k):
    counts = Counter(nums)                  # value → how many times
    # nlargest gives the k items with the biggest count
    return [val for val, _ in heapq.nlargest(k, counts.items(), key=lambda p:
p[1])]
```

JAVA

```java
int[] topKFrequent(int[] nums, int k) {
    Map<Integer, Integer> counts = new HashMap<>();
    for (int x : nums) counts.merge(x, 1, Integer::sum);
    // min-heap ordered by frequency, keep only k biggest
    PriorityQueue<int[]> heap =
        new PriorityQueue<>((a, b) -> a[1] - b[1]);
    for (Map.Entry<Integer, Integer> e : counts.entrySet()) {
        heap.offer(new int[]{ e.getKey(), e.getValue() });
        if (heap.size() > k) heap.poll();
    }
    int[] result = new int[k];
    for (int i = 0; i < k; i++) result[i] = heap.poll()[0];
    return result;
}
```

MIN-HEAP FOR THE LARGEST, MAX-HEAP FOR THE SMALLEST

It feels backwards but it is the key insight for "top K" problems. To keep the K largest items, use a min-heap of size K and pop the smallest whenever it overflows. The heap top is the weakest survivor, so anything weaker gets thrown out. Flip the logic for the K smallest.

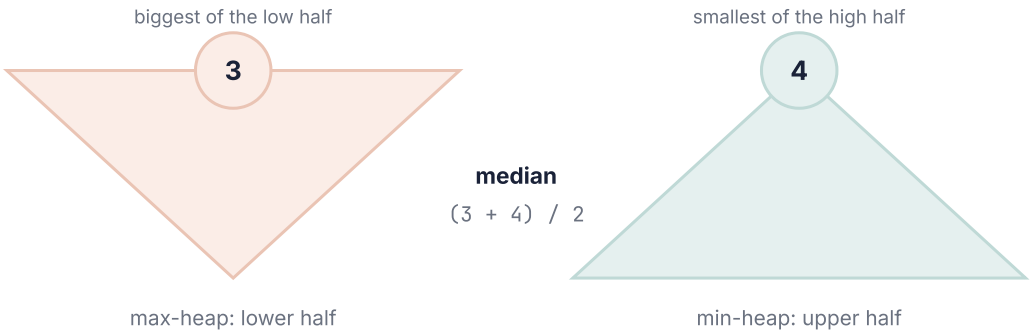
WATCH OUT

A heap is not sorted. It only guarantees the top element is the min or max. If you iterate over a heap's internal array you will not get sorted order. Also, in Python remember to negate values for a max-heap and negate again when you read them back, which is an easy place to introduce a sign bug.

DEEPER INTUITION

Why two heaps give a running median

A heap only promises you the single extreme element, the smallest or the largest, in O(1), while pushes and pops cost O(log n). For a running median you keep two heaps: a max-heap holding the smaller half of the numbers and a min-heap holding the larger half, kept balanced in size. The median is then always sitting right at their tops, either the top of the bigger heap or the average of the two tops.



Keep the smaller half in a max-heap and the larger half in a min-heap. The median sits at the two tops.

Another worked example: Find Median from Data Stream

Numbers arrive one at a time and you must report the median at any point. Push each number so the low half stays in a max-heap and the high half in a min-heap, rebalancing so their sizes differ by at most one. The median is read straight off the tops.

PYTHON

```python
import heapq

class MedianFinder:
    def __init__(self):
        self.low = []    # max-heap via negatives, the smaller half
        self.high = []   # min-heap, the larger half

    def add_num(self, num):
        heapq.heappush(self.low, -num)
        heapq.heappush(self.high, -heapq.heappop(self.low))   # balance value
        if len(self.high) > len(self.low):                    # balance size
            heapq.heappush(self.low, -heapq.heappop(self.high))

    def find_median(self):
        if len(self.low) > len(self.high):
            return -self.low[0]
        return (-self.low[0] + self.high[0]) / 2
```

JAVA

```java
class MedianFinder {
    PriorityQueue<Integer> low = new PriorityQueue<>
(Collections.reverseOrder());
    PriorityQueue<Integer> high = new PriorityQueue<>();

    void addNum(int num) {
        low.offer(num);
        high.offer(low.poll());          // balance value
        if (high.size() > low.size())     // balance size
            low.offer(high.poll());
    }
    double findMedian() {
        if (low.size() > high.size()) return low.peek();
        return (low.peek() + high.peek()) / 2.0;
    }
}
```

GOING DEEPER

What kinds of problems this solves

USE THIS PATTERN FOR

Whenever you repeatedly need the smallest or largest item without fully sorting: top K, the kth largest, a running median, merging many sorted streams, or scheduling by priority.

Problem type	Classic examples
Top K and Kth	Kth Largest Element in an Array, Top K Frequent Elements, K Closest Points to Origin, Kth Largest in a Stream
Merge and schedule	Merge k Sorted Lists, Task Scheduler, Reorganize String, Meeting Rooms II
Two heaps	Find Median from Data Stream, Sliding Window Median, IPO

The algorithm, in a bit more detail

A heap keeps the extreme element (smallest for a min heap) always at the top, ready in O(1), while pushing and popping cost only O(log n). It does not keep everything sorted, it just guarantees the top, which is exactly what you want when you only ever need the current best.

The key pattern for top K is a heap of size k. Keep pushing, and whenever the heap grows past k, pop the worst. What remains is the k best, computed in O(n log k) instead of sorting everything in O(n log n). For a running median, hold a max heap of the lower half and a min heap of the upper half, and keep their sizes balanced so the median sits at the tops.

Variations you will run into

You will use a **min heap**, a **max heap** (in Python, push negatives since it only has a min heap), a **fixed size k heap** for top K, and the **two heap** setup for medians. Storing tuples like (priority, item) lets the heap order by the first field.

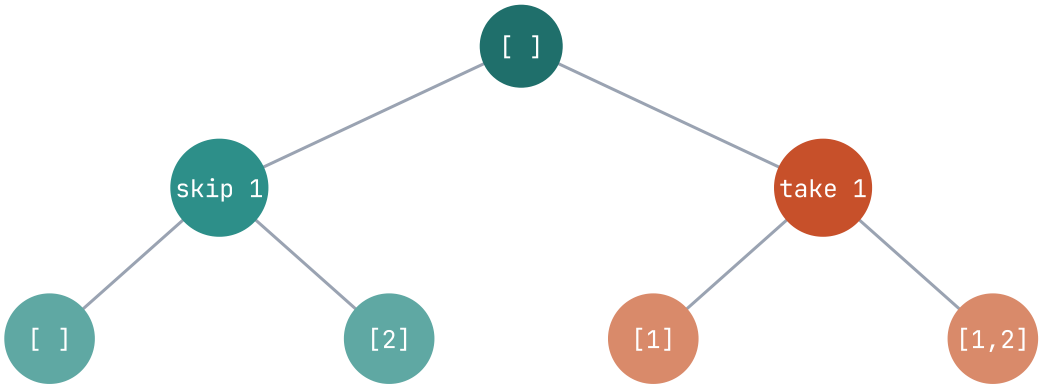
Edge cases and gotchas

- Python heapq is a min heap only, so negate values or use a key wrapper to get max heap behavior.
- When you push tuples, make sure the tie breaking field is comparable, or add a counter to avoid comparing raw objects.
- Building a heap from a list with heapify is O(n), which is cheaper than pushing one by one.
- For top K largest, a min heap of size k is correct, since you pop the smallest to keep the largest k.

RECURSION FAMILY

11 Backtracking

Backtracking is organized trial and error. You build a solution one choice at a time. When a choice leads nowhere, you undo it and try the next option. It is how you generate all subsets, all permutations, all combinations, and how you solve puzzles like Sudoku and the N-Queens board.



at each item you branch: include it or skip it

the leaves are all the subsets

Backtracking explores this tree of choices, then rewinds to try the other branch.

REACH FOR BACKTRACKING WHEN YOU SEE

"Generate all...", "find every combination / permutation / subset", "list all valid arrangements", or a constraint puzzle. The giveaway in the constraints is a small input size, often n up to about 15 or 20, because the number of possibilities grows explosively and only stays feasible when n is small.

The universal template: choose, explore, unchoose

Every backtracking solution has the same rhythm. You make a choice (add something to your current path), you recurse to build on it, then you undo the choice (remove it) so you can try the next option. That undo step is the "backtrack," and it is what makes the whole search work.

PYTHON

```python
def backtrack(path, options):
    if is_complete(path):        # reached a full solution
        record(path)
        return
    for choice in options:
        path.append(choice)      # choose
        backtrack(path, next_options(choice))   # explore
        path.pop()               # unchoose (this is the backtrack)
```

JAVA

```java
void backtrack(List<Integer> path, ...) {
    if (isComplete(path)) {      // full solution reached
        record(path);
        return;
    }
    for (int choice : options) {
        path.add(choice);        // choose
        backtrack(path, ...);    // explore
        path.remove(path.size() - 1);  // unchoose
    }
}
```

Worked example: all subsets

For each element you have two choices: include it or leave it out. Recurse through the elements, and every time you run out of elements you have formed one complete subset. Record a copy of the current path at every node, because subsets include the partial ones too.

PYTHON

```python
def subsets(nums):
    result = []
    def backtrack(start, path):
        result.append(path[:])          # record a copy of the current subset
        for i in range(start, len(nums)):
            path.append(nums[i])        # choose nums[i]
            backtrack(i + 1, path)      # explore with the rest
            path.pop()                  # unchoose
    backtrack(0, [])
    return result
```

JAVA

```java
List<List<Integer>> subsets(int[] nums) {
    List<List<Integer>> result = new ArrayList<>();
    backtrack(nums, 0, new ArrayList<>(), result);
    return result;
}
void backtrack(int[] nums, int start, List<Integer> path,
               List<List<Integer>> result) {
    result.add(new ArrayList<>(path));   // record a copy
    for (int i = start; i < nums.length; i++) {
        path.add(nums[i]);               // choose
        backtrack(nums, i + 1, path, result);  // explore
        path.remove(path.size() - 1);    // unchoose
    }
}
```

Worked example: all permutations

Permutations differ from subsets in one way: order matters and you use every element. So instead of a start index, you track which elements are still available and pick any unused one at each step.

PYTHON

```python
def permutations(nums):
    result = []
    used = [False] * len(nums)
    def backtrack(path):
        if len(path) == len(nums):      # used everything
            result.append(path[:])
            return
        for i in range(len(nums)):
            if used[i]:
                continue                # skip already-used elements
            used[i] = True
            path.append(nums[i])        # choose
            backtrack(path)             # explore
            path.pop()                  # unchoose
            used[i] = False
    backtrack([])
    return result
```

JAVA

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

```
    }
}
```

PRUNE EARLY TO GO FASTER

The real speed in backtracking comes from cutting off dead branches as soon as you can tell they will fail. In combination sum, if the running total already passed the target, stop; do not keep adding. In N-Queens, check for conflicts before placing a queen, not after. A good prune can turn an impossible search into an instant one.

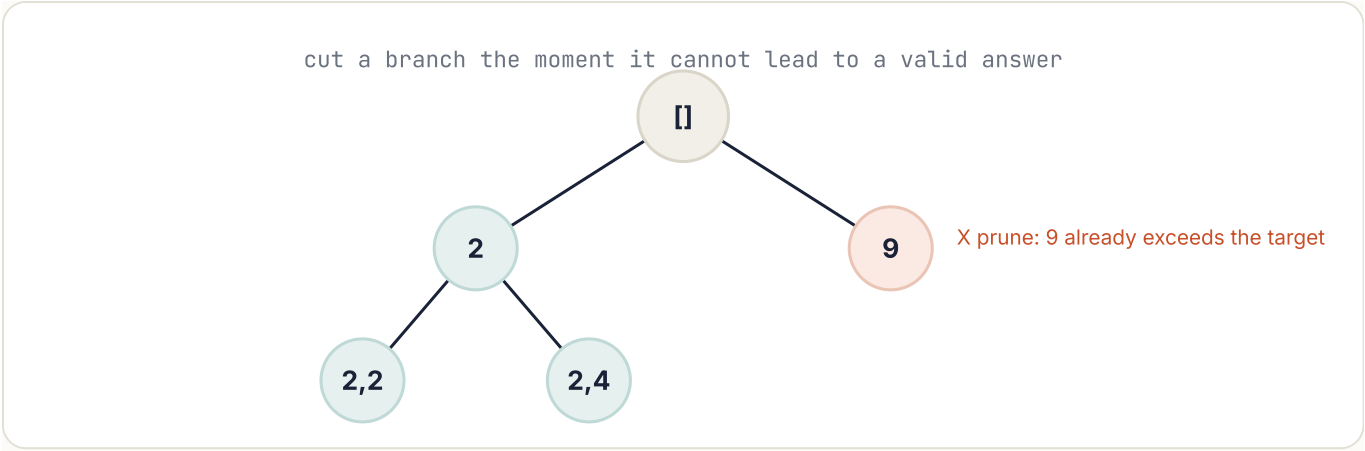
WATCH OUT

When you save a solution, save a copy, not the live path. In Python that is `path[:]`, in Java it is `new ArrayList<>(path)`. If you store the path directly, later backtracking edits it and every saved answer ends up identical and wrong. This single mistake accounts for a huge share of backtracking bugs.

DEEPER INTUITION

Why choose, explore, unchoose works

Backtracking is depth first search over the tree of all choices. You choose an option, recurse to explore everything that follows, then undo the choice so you can try the next one. That rhythm visits every valid arrangement exactly once. The thing that keeps it from being hopelessly slow is pruning: the instant a partial choice cannot lead to a valid answer, you stop and back out instead of exploring a dead branch.



Backtracking explores the tree of choices, but pruning skips whole branches that cannot possibly work.

Another worked example: Combination Sum

Find every combination of numbers that adds up to a target, where each number may be reused. Try candidates from a start index so you never repeat a combination in a different order, and prune any branch whose next candidate already exceeds what remains.

PYTHON

```python
def combination_sum(candidates, target):
    result = []
    def backtrack(start, remaining, path):
        if remaining == 0:
            result.append(path[:])         # a valid combination
            return
        for i in range(start, len(candidates)):
            if candidates[i] > remaining:
                continue                   # prune: too big
            path.append(candidates[i])
            backtrack(i, remaining - candidates[i], path)  # reuse allowed
            path.pop()                     # unchoose
    backtrack(0, target, [])
    return result
```

JAVA

```java
List<List<Integer>> combinationSum(int[] candidates, int target) {
    List<List<Integer>> result = new ArrayList<>();
    backtrack(candidates, 0, target, new ArrayList<>(), result);
    return result;
}
void backtrack(int[] c, int start, int remaining,
               List<Integer> path, List<List<Integer>> result) {
    if (remaining == 0) { result.add(new ArrayList<>(path)); return; }
    for (int i = start; i < c.length; i++) {
        if (c[i] > remaining) continue;           // prune
        path.add(c[i]);
        backtrack(c, i, remaining - c[i], path, result);  // reuse i
        path.remove(path.size() - 1);             // unchoose
    }
}
```

GOING DEEPER

What kinds of problems this solves

USE THIS PATTERN FOR

Any request to build all of something by making a sequence of choices: every subset, every permutation, every combination, or filling a board under constraints. If the word all appears and the input is small, this is usually it.

Problem type	Classic examples
Subsets and combinations	Subsets, Subsets II, Combinations, Combination Sum, Combination Sum II, Letter Combinations of a Phone Number
Permutations	Permutations, Permutations II, Next Permutation via ordering
Partitioning and boards	Palindrome Partitioning, N-Queens, Sudoku Solver, Word Search, Restore IP Addresses

The algorithm, in a bit more detail

Backtracking is depth first search over the tree of choices. At each step you choose an option, recurse to explore everything that follows from it, then undo the choice so you can try the next option. That choose, explore, unchoose rhythm is the whole template, and it guarantees you visit every valid arrangement exactly once.

The thing that keeps it from being hopelessly slow is pruning. The moment a partial choice cannot possibly lead to a valid answer, you stop and back out instead of exploring a dead branch. Good pruning is often the difference between passing and timing out on N-Queens or Sudoku.

Variations you will run into

For **combinations** you pass a start index so you never reuse earlier elements and never repeat a set in a different order. For **permutations** you track which elements are already used. To handle duplicates, sort first and skip a value that equals the previous one at the same depth.

Edge cases and gotchas

- Always undo your choice after the recursive call, or state leaks between branches.
- When you record a solution, add a copy of the current path, not the live list, since it keeps changing.
- Sort and skip duplicates to avoid emitting the same subset or permutation twice.
- Prune early with a feasibility check, otherwise the exponential blowup will time out.

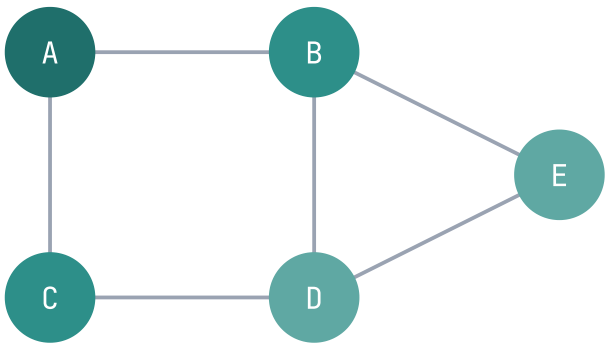
STEP BY STEP

Generating all subsets of [1, 2] with choose, explore, unchoose. Every path down the decision tree records one subset.

1. Start with the empty subset [] and record it.
2. Choose 1, giving [1], and record it.
3. From [1], choose 2, giving [1, 2], and record it, then unchoose 2 to return to [1].
4. Unchoose 1 to return to []. Now choose 2, giving [2], and record it, then unchoose 2.
5. Every choice has been explored. Subsets found: [], [1], [1, 2], [2].

12 Graphs, BFS, and DFS

A graph is nodes connected by edges. Roads between cities, friends on a network, rooms with doors between them. Trees are just a special kind of graph. Two traversals do most of the work: breadth first (explore in rings outward) and depth first (dive down one path). Master these two and a whole category of problems opens up.



BFS from A:
A, B, C, D, E

DFS from A:
A, B, D, C, E

(exact order depends on neighbor order)

BFS spreads out level by level. DFS commits to one path before backing up.

How to store a graph

The usual representation is an adjacency list: a map from each node to the list of its neighbors. It is compact and fast to iterate. For grid problems (mazes, islands), the grid itself is the graph, and a cell's neighbors are the cells up, down, left, and right.

REACH FOR GRAPH TRAVERSAL WHEN YOU SEE

"Number of islands / connected groups", "can you reach B from A", "shortest path in an unweighted grid or graph" (BFS), "does a cycle exist", "course prerequisites / build order" (topological sort). Grids of land and water, rooms and keys, and word ladders are all graphs in disguise.

BFS: shortest steps in an unweighted graph

BFS uses a queue and visits nodes in rings: all nodes one step away, then all two steps away, and so on. Because it reaches nodes in order of distance, the first time it touches the target is guaranteed to be the shortest path. Always mark nodes as visited when you enqueue them, so you never add the same node twice.

PYTHON

```python
from collections import deque

def bfs(graph, start):
    visited = {start}
    queue = deque([start])
    order = []
    while queue:
        node = queue.popleft()
        order.append(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)     # mark on enqueue, not on dequeue
                queue.append(neighbor)
    return order
```

JAVA

```java
List<Integer> bfs(Map<Integer, List<Integer>> graph, int start) {
    Set<Integer> visited = new HashSet<>();
    Queue<Integer> queue = new LinkedList<>();
    List<Integer> order = new ArrayList<>();
    visited.add(start);
    queue.offer(start);
    while (!queue.isEmpty()) {
        int node = queue.poll();
        order.add(node);
        for (int neighbor : graph.getOrDefault(node, List.of())) {
            if (!visited.contains(neighbor)) {
                visited.add(neighbor);    // mark on enqueue
                queue.offer(neighbor);
            }
        }
    }
    return order;
}
```

DFS: explore everything reachable

DFS dives down one branch as far as it goes, then backtracks. It can be written with recursion (simplest) or with an explicit stack. Use it to count connected components, check reachability, or explore all paths.

PYTHON

```python
def dfs(graph, node, visited):
    visited.add(node)
    for neighbor in graph[node]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)
```

JAVA

```java
void dfs(Map<Integer, List<Integer>> graph, int node, Set<Integer> visited) {
    visited.add(node);
    for (int neighbor : graph.getOrDefault(node, List.of())) {
        if (!visited.contains(neighbor)) {
            dfs(graph, neighbor, visited);
        }
    }
}
```

Worked example: number of islands

Given a grid of land ('1') and water ('0'), count the islands. Scan the grid. Every time you hit unvisited land, you found a new island, so add one and then flood the whole island by DFS, marking every connected land cell so you do not count it again.

PYTHON

```python
def num_islands(grid):
    if not grid:
        return 0
    rows, cols = len(grid), len(grid[0])
    count = 0

    def sink(r, c):
        if r < 0 or r ≥ rows or c < 0 or c ≥ cols or grid[r][c] ≠ '1':
            return
        grid[r][c] = '0'          # mark as visited by sinking it
        sink(r + 1, c); sink(r - 1, c)
        sink(r, c + 1); sink(r, c - 1)

    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == '1':
                count += 1        # a new island
                sink(r, c)        # flood it so we do not recount
    return count
```

JAVA

```java
int numIslands(char[][] grid) {
    if (grid.length == 0) return 0;
    int rows = grid.length, cols = grid[0].length, count = 0;
    for (int r = 0; r < rows; r++) {
        for (int c = 0; c < cols; c++) {
            if (grid[r][c] == '1') {
                count++;          // a new island
                sink(grid, r, c); // flood it
            }
        }
    }
    return count;
}
void sink(char[][] grid, int r, int c) {
    if (r < 0 || r ≥ grid.length || c < 0 || c ≥ grid[0].length
        || grid[r][c] ≠ '1') return;
    grid[r][c] = '0';             // mark visited
    sink(grid, r + 1, c); sink(grid, r - 1, c);
```

```
    sink(grid, r, c + 1); sink(grid, r, c - 1);
}
```

Topological sort: ordering with prerequisites

When tasks depend on each other, like courses that require other courses first, you need an order where every dependency comes before the thing that needs it. Count how many prerequisites each task has (its in-degree). Start with the tasks that have none, and each time you finish one, reduce the count of everything that depended on it. When a count hits zero, that task is ready. If you cannot finish everything, there was a cycle.

PYTHON

```python
from collections import deque, defaultdict

def can_finish(num_courses, prerequisites):
    graph = defaultdict(list)
    indegree = [0] * num_courses
    for course, need in prerequisites:
        graph[need].append(course)
        indegree[course] += 1

    queue = deque(c for c in range(num_courses) if indegree[c] == 0)
    done = 0
    while queue:
        course = queue.popleft()
        done += 1
        for nxt in graph[course]:
            indegree[nxt] -= 1        # one prerequisite satisfied
            if indegree[nxt] == 0:
                queue.append(nxt)     # now ready
    return done == num_courses        # false means a cycle blocked us
```

JAVA

```java
boolean canFinish(int numCourses, int[][] prerequisites) {
    List<List<Integer>> graph = new ArrayList<>();
    for (int i = 0; i < numCourses; i++) graph.add(new ArrayList<>());
    int[] indegree = new int[numCourses];
    for (int[] p : prerequisites) {
        graph.get(p[1]).add(p[0]);
        indegree[p[0]]++;
    }
    Queue<Integer> queue = new LinkedList<>();
    for (int c = 0; c < numCourses; c++)
        if (indegree[c] == 0) queue.offer(c);
    int done = 0;
    while (!queue.isEmpty()) {
        int course = queue.poll();
        done++;
        for (int nxt : graph.get(course)) {
            if (--indegree[nxt] == 0) queue.offer(nxt);
        }
```

```
    }
    return done == numCourses;   // false → cycle
}
```

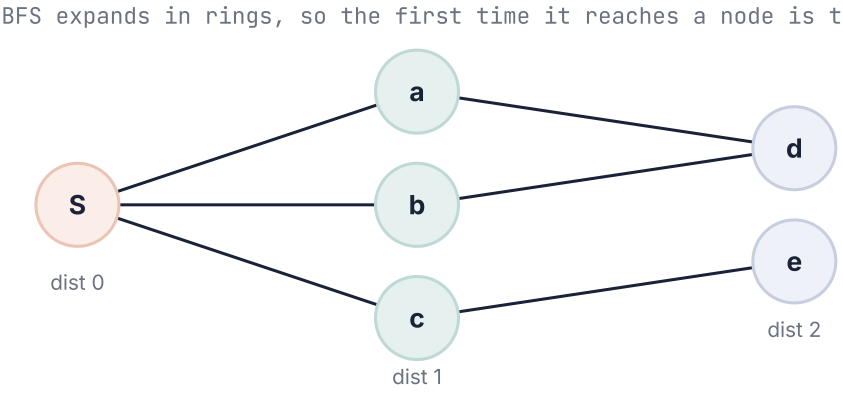
WATCH OUT

Forgetting to mark nodes visited is the classic graph bug; it causes infinite loops or exponential blowups. In BFS, mark when you enqueue, not when you dequeue, or the same node gets added many times. For shortest path in an unweighted graph use BFS, not DFS, because DFS does not find shortest paths. And if edges have weights, you need Dijkstra's algorithm, which is BFS with a heap instead of a plain queue.

DEEPER INTUITION

Why BFS finds shortest paths

Breadth first search explores in rings using a queue: everything one step from the source, then everything two steps away, and so on. Because it reaches nodes in order of distance, the very first time it touches a node it has arrived by a shortest path, which is exactly why BFS answers shortest path questions on unweighted graphs. Depth first search instead dives as far as it can, which is perfect for flooding a whole region or detecting a cycle.



Breadth first search visits everything one step away, then two steps, and so on, which is why it finds shortest paths in an unweighted graph.

Another worked example: Rotting Oranges

Rotten oranges infect fresh neighbors each minute. Ask how many minutes until none are fresh. This is multi source BFS: start with every rotten orange in the queue at once, and each BFS level is one minute of spreading. If any fresh orange is unreachable, return minus one.

PYTHON

```python
from collections import deque

def oranges_rotting(grid):
    rows, cols = len(grid), len(grid[0])
    queue, fresh = deque(), 0
    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == 2: queue.append((r, c))   # all rotten start now
            elif grid[r][c] == 1: fresh += 1
    minutes = 0
    while queue and fresh:
        minutes += 1
        for _ in range(len(queue)):                    # one level = one
minute
            r, c = queue.popleft()
            for dr, dc in ((1,0),(-1,0),(0,1),(0,-1)):
                nr, nc = r + dr, c + dc
                if 0 <= nr < rows and 0 <= nc < cols and grid[nr][nc] == 1:
                    grid[nr][nc] = 2
                    fresh -= 1
                    queue.append((nr, nc))
    return -1 if fresh else minutes
```

JAVA

```java
int orangesRotting(int[][] grid) {
    int rows = grid.length, cols = grid[0].length, fresh = 0;
    Queue<int[]> queue = new ArrayDeque<>();
    for (int r = 0; r < rows; r++)
        for (int c = 0; c < cols; c++) {
            if (grid[r][c] == 2) queue.offer(new int[]{r, c});
            else if (grid[r][c] == 1) fresh++;
        }
    int minutes = 0;
    int[][] dirs = {{1,0},{-1,0},{0,1},{0,-1}};
    while (!queue.isEmpty() && fresh > 0) {
        minutes++;
        for (int n = queue.size(); n > 0; n--) {
            int[] cell = queue.poll();
            for (int[] d : dirs) {
```

```
                int nr = cell[0] + d[0], nc = cell[1] + d[1];
                if (nr >= 0 && nr < rows && nc >= 0 && nc < cols && grid[nr]
[nc] == 1) {
                    grid[nr][nc] = 2; fresh--;
                    queue.offer(new int[]{nr, nc});
                }
            }
        }
    }
    return fresh == 0 ? minutes : -1;
}
```

GOING DEEPER

What kinds of problems this solves

USE THIS PATTERN FOR

Anything with things connected to other things: grids and islands, reachability and connected groups, shortest path when every step costs the same, ordering tasks with prerequisites, and detecting cycles.

Problem type	Classic examples
Grid and flood fill	Number of Islands, Max Area of Island, Rotting Oranges, Flood Fill, Surrounded Regions, Walls and Gates
Connectivity and traversal	Clone Graph, Number of Connected Components, Course Schedule, Pacific Atlantic Water Flow
Topological sort	Course Schedule II, Alien Dictionary, Minimum Height Trees
Shortest path with BFS	Word Ladder, Shortest Path in Binary Matrix, Open the Lock, Rotting Oranges

The algorithm, in a bit more detail

Two engines cover most of this chapter. Breadth first search explores in rings using a queue, so the first time it reaches a node it has found the shortest path in an unweighted graph. Depth first search dives as far as it can using recursion or a stack, which is perfect for exploring a whole region, counting components, or detecting a cycle.

Topological sort orders tasks so every prerequisite comes before what depends on it. You count incoming edges for each node, start from the ones with none, and peel them off, reducing counts as you go. If you cannot peel everything, there is a cycle, which is exactly how "can you finish all courses" is answered.

Variations you will run into

Grid problems treat each cell as a node with up to four neighbors. **Multi source BFS** starts from many cells at once, which is how rotting oranges spreads. When edges have weights you move up to **Dijkstra** with a heap, and for pure connectivity questions **union find** is often the cleanest tool.

Edge cases and gotchas

- Mark a node visited when you enqueue it in BFS, not when you dequeue it, or you will process it many times.
- Be clear whether the graph is directed or undirected, since it changes cycle detection and edge building.
- For grids, check bounds before you step, and remember diagonal moves only count if the problem says so.
- Topological sort only works on a directed acyclic graph, and a leftover node means a cycle exists.

STEP BY STEP

Counting islands in a tiny grid where 1 is land and 0 is water. Row one is 1 1 0 and row two is 0 1 0. We scan for unvisited land and flood each island we find.

1. Scan cells left to right, top to bottom, looking for a 1 that has not been visited.
2. Find land at the top left. Start island number 1 and begin a flood fill from there.

3 The flood spreads to the connected land: the top left, its right neighbor, and the cell below that. Mark all three as visited.

4 Keep scanning. Every remaining cell is water or already visited, so no new island starts.

5 Total islands found: 1.

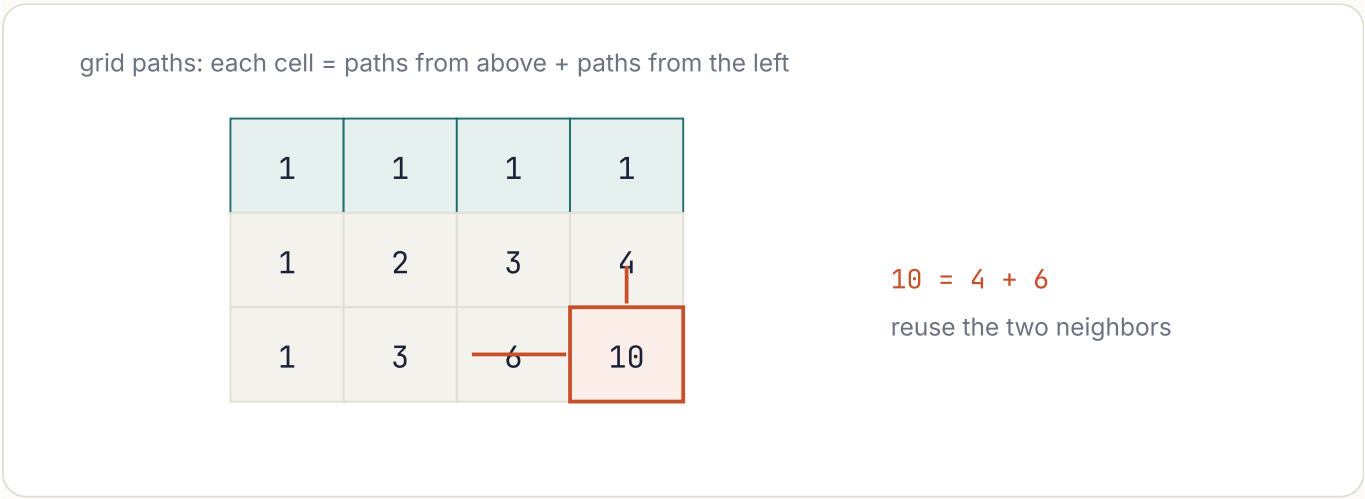
13

Dynamic programming

Dynamic programming, or DP, sounds scary but it is one idea: break a problem into smaller versions of itself, and never solve the same small version twice. You solve each subproblem once, write down the answer, and reuse it. That is the whole trick.

How to know it is a DP problem

Two signals together mean DP. First, the problem asks for a best value ("minimum cost", "maximum profit", "longest", "number of ways"). Second, the answer to the whole problem can be built from answers to smaller pieces, and those pieces overlap so you keep recomputing them. When you see "how many ways" or "min / max" and a natural notion of a smaller subproblem, think DP.



Fill the table so that when you compute a cell, the cells it depends on are already done.

Two ways to write DP

Top down (memoization). Write the natural recursion, then cache each answer in a map or array so repeated calls are instant. Easiest to think about because it mirrors the problem directly.

Bottom up (tabulation). Fill a table starting from the smallest subproblems and building up to the answer. Usually a bit faster and avoids deep recursion, but you have to figure out the fill order.

THE FOUR QUESTIONS THAT CRACK ANY DP

1. What is the state? (What does one subproblem look like, and what index or indices describe it?) 2. What is the recurrence? (How do you build this state from smaller states?) 3. What are the base cases? (The smallest states you know directly.) 4. What is the answer? (Which state holds the final result?) Answer these four and the code writes itself.

Worked example: climbing stairs (the "hello world" of DP)

You can climb 1 or 2 steps at a time. How many ways to reach step n? To reach step n, your last move was either from step n-1 or from step n-2, so the ways to reach n equal the ways to reach n-1 plus the ways to reach n-2. That is the Fibonacci sequence.

PYTHON

```python
def climb_stairs(n):
    if n <= 2:
        return n
    # only the last two results matter, so keep just those
    one_back, two_back = 2, 1
    for _ in range(3, n + 1):
        current = one_back + two_back
        two_back = one_back
        one_back = current
    return one_back
```

JAVA

```java
int climbStairs(int n) {
    if (n <= 2) return n;
    int oneBack = 2, twoBack = 1;   // ways to reach n-1 and n-2
    for (int i = 3; i <= n; i++) {
        int current = oneBack + twoBack;
        twoBack = oneBack;
        oneBack = current;
    }
    return oneBack;
}
```

ROLLING VARIABLES SAVE MEMORY

When each state only depends on the previous one or two, you do not need a whole array. Keep just the last couple of values in variables and slide them forward. Climbing stairs drops from O(n) space to O(1) space this way. Many 1D DP problems allow the same trick.

Worked example: coin change (minimum coins)

Given coin denominations and an amount, find the fewest coins that make that amount. The state is "fewest coins to make amount a." To make amount a, try each coin: use it, then you still need to make a minus that coin's value, which is a smaller subproblem you already solved. Take the best over all coins.

PYTHON

```python
def coin_change(coins, amount):
    INF = amount + 1
    dp = [0] + [INF] * amount     # dp[a] = fewest coins to make a
    for a in range(1, amount + 1):
        for coin in coins:
            if coin ≤ a:
                dp[a] = min(dp[a], dp[a - coin] + 1)
    return dp[amount] if dp[amount] ≠ INF else -1
```

JAVA

```java
int coinChange(int[] coins, int amount) {
    int INF = amount + 1;
    int[] dp = new int[amount + 1];
    Arrays.fill(dp, INF);
    dp[0] = 0;                        // zero coins make amount 0
    for (int a = 1; a ≤ amount; a++) {
        for (int coin : coins) {
            if (coin ≤ a) dp[a] = Math.min(dp[a], dp[a - coin] + 1);
        }
    }
    return dp[amount] == INF ? -1 : dp[amount];
}
```

Worked example: longest common subsequence (2D DP)

Given two strings, find the length of the longest sequence of characters that appears in both, in order but not necessarily together. The state is a pair: how far you are into each string. If the current characters match, they extend the best answer from one step back in both strings. If not, you take the better of skipping a character from either string. This grid-filling shape appears in edit distance, string matching, and many more.

PYTHON

```python
def longest_common_subseq(a, b):
    m, n = len(a), len(b)
    dp = [[0] * (n + 1) for _ in range(m + 1)]
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if a[i - 1] == b[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1   # characters match
            else:
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])  # skip one
    return dp[m][n]
```

JAVA

```java
int longestCommonSubseq(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++) {
            if (a.charAt(i - 1) == b.charAt(j - 1)) {
                dp[i][j] = dp[i - 1][j - 1] + 1;        // match
            } else {
                dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }
    return dp[m][n];
}
```

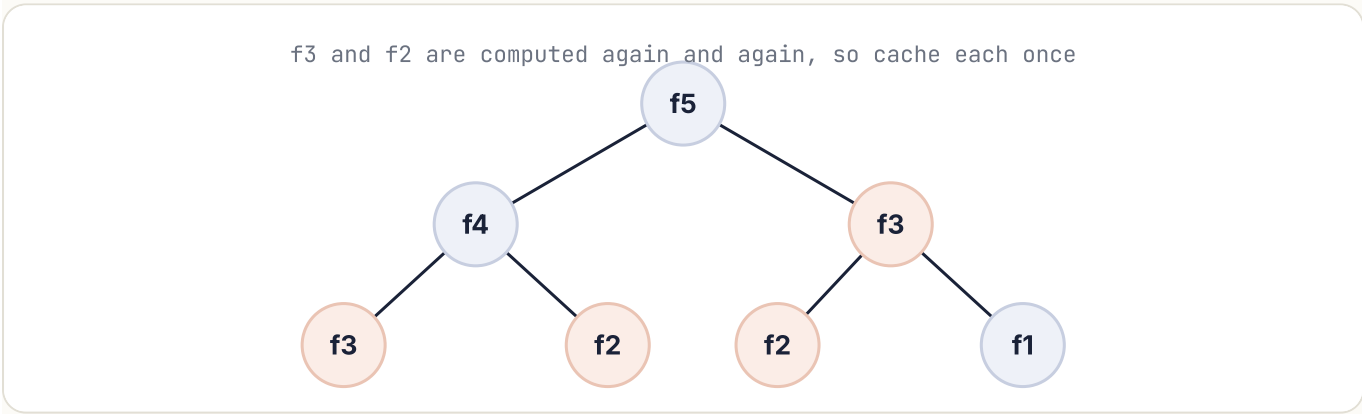
WATCH OUT

Get the base cases right or everything downstream is wrong; an off-by-one in the table size or the starting values is the usual culprit. Make sure you fill the table in an order where every cell's dependencies are already computed. And do not force tabulation if the recursion is clearer to you; a memoized top-down solution earns full marks and is often easier to get correct under pressure.

DEEPER INTUITION

Why memoizing turns exponential into linear

Dynamic programming applies when a plain recursion solves the same smaller problem over and over. The recursion tree for something like Fibonacci is full of repeated calls, so its cost blows up exponentially. If you remember each subproblem's answer the first time you compute it, every later call is a free lookup, and the work collapses to the number of distinct subproblems. That is the whole idea, whether you write it top down with a cache or bottom up as a table.



A naive recursion recomputes the same subproblems (the orange nodes). Memoizing each result once removes the waste.

Another worked example: House Robber

You cannot rob two adjacent houses. The best you can do at each house is either skip it, keeping the previous best, or rob it and add its money to the best from two houses back. Track just those two rolling values, so it runs in O(n) time and O(1) space.

PYTHON

```python
def rob(nums):
    prev, curr = 0, 0               # best up to the last two houses
    for money in nums:
        prev, curr = curr, max(curr, prev + money)   # skip vs rob this house
    return curr
```

JAVA

```java
int rob(int[] nums) {
    int prev = 0, curr = 0;
    for (int money : nums) {
        int take = prev + money;        // rob this house
        prev = curr;
        curr = Math.max(curr, take);    // skip vs rob
    }
    return curr;
}
```

GOING DEEPER

What kinds of problems this solves

USE THIS PATTERN FOR

Problems asking for a count of ways, or a best possible value, where the same smaller problems keep reappearing. If a brute force recursion solves the same subproblem over and over, dynamic programming remembers each answer once.

Problem type	Classic examples
One dimensional sequence	Climbing Stairs, House Robber, House Robber II, Decode Ways, Maximum Subarray, Word Break
Knapsack and coins	Coin Change, Coin Change II, Partition Equal Subset Sum, Target Sum
Grid paths	Unique Paths, Minimum Path Sum, Maximal Square, Dungeon Game
Two sequences and strings	Longest Common Subsequence, Edit Distance, Longest Palindromic Substring, Distinct Subsequences
Choice over positions	Longest Increasing Subsequence, Best Time to Buy and Sell Stock with Cooldown, Burst Balloons

The algorithm, in a bit more detail

Every DP is four questions. What is the state, meaning the smallest set of facts that describe a subproblem. What is the recurrence, meaning how a state is built from smaller states. What are the base cases, the smallest states you can fill directly. And what order fills the table so every state is ready before you need it.

You can write it top down as memoized recursion, where you solve naturally and cache each answer, or bottom up as a table you fill in order. They compute the same thing. Top down is often easier to reason about, bottom up is often faster and lets you shrink memory by keeping only the last row or two.

Variations you will run into

The families to recognize are **linear DP** over one sequence, **knapsack** style choices of take or skip, **grid DP** over a 2D board, and **two sequence DP** for string alignment like edit distance. Many one dimensional DPs can drop to O(1) extra space by keeping just the previous couple of values.

Edge cases and gotchas

- Define the state precisely, since a vague state is the number one reason a DP is wrong.

- Get the base cases right, they anchor everything built on top of them.
- Fill the table in an order where every value you read is already computed.
- Watch array sizes and off by one, especially when the state runs from zero to n inclusive.

STEP BY STEP

Climbing stairs for 5 steps, where you can take 1 or 2 at a time. The ways to reach a step are the ways to reach the two steps below it, so we build a small table.

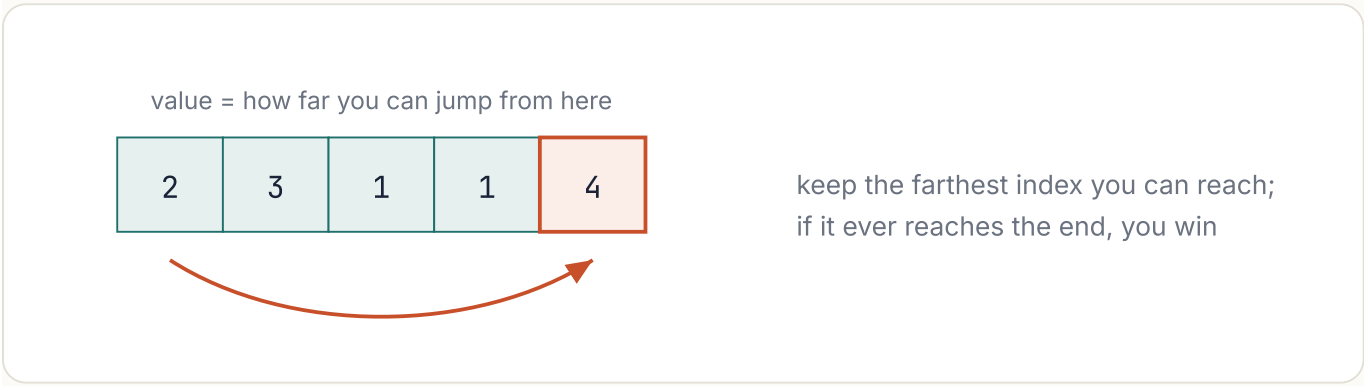
step n	ways to reach it	how
1	1	`base case`
2	2	`base case`
3	3	`ways(2) + ways(1) = 2 + 1`
4	5	`ways(3) + ways(2) = 3 + 2`
5	8	`ways(4) + ways(3) = 5 + 3`

14 Greedy

A greedy algorithm makes the choice that looks best right now and never reconsiders. When that works, it is wonderfully simple and fast. The catch is that grabbing the locally best option does not always lead to the globally best result, so the skill is knowing when greedy is safe.

Greedy versus dynamic programming

Both break a problem into steps. The difference: DP considers all options at each step and remembers them; greedy commits to one option and moves on. Greedy is correct only when a locally optimal choice provably leads to a globally optimal answer. If you are not sure it does, fall back to DP, which is safer but slower. On LeetCode, if a greedy idea passes your hand-checked examples and you can argue why the greedy choice never hurts, trust it.



Jump game: one pass, always tracking the farthest reachable point.

REACH FOR GREEDY WHEN YOU SEE

"Maximum number of non-overlapping intervals", "minimum number of X to cover Y", "can you reach the end", "best time to buy and sell". Often a sort comes first (by end time, by size, by value), and then a single greedy pass does the rest.

Worked example: jump game

Each number tells you the maximum you can jump from that spot. Can you reach the last index? Track the farthest index reachable so far. Walk forward; if you ever stand on a spot beyond your

farthest reach, you are stuck. Otherwise extend the reach. If the reach ever covers the last index, you made it.

PYTHON

```python
def can_jump(nums):
    farthest = 0
    for i, jump in enumerate(nums):
        if i > farthest:
            return False          # this spot is unreachable
        farthest = max(farthest, i + jump)
        if farthest >= len(nums) - 1:
            return True           # can already reach the end
    return True
```

JAVA

```java
boolean canJump(int[] nums) {
    int farthest = 0;
    for (int i = 0; i < nums.length; i++) {
        if (i > farthest) return false;        // unreachable
        farthest = Math.max(farthest, i + nums[i]);
        if (farthest >= nums.length - 1) return true;
    }
    return true;
}
```

Worked example: best time to buy and sell stock

Given daily prices, find the best single buy-then-sell profit. The greedy insight: to sell on any given day for the most profit, you want to have bought at the lowest price seen up to that day. So track the minimum price so far, and at each day check the profit if you sold today. One pass, no DP table needed.

PYTHON

```python
def max_profit(prices):
    min_price = float('inf')
    best = 0
    for price in prices:
        min_price = min(min_price, price)     # cheapest day so far
        best = max(best, price - min_price)   # profit if we sell today
    return best
```

JAVA

```java
int maxProfit(int[] prices) {
    int minPrice = Integer.MAX_VALUE, best = 0;
    for (int price : prices) {
        minPrice = Math.min(minPrice, price);      // cheapest so far
        best = Math.max(best, price - minPrice);   // sell today?
    }
    return best;
}
```

WATCH OUT

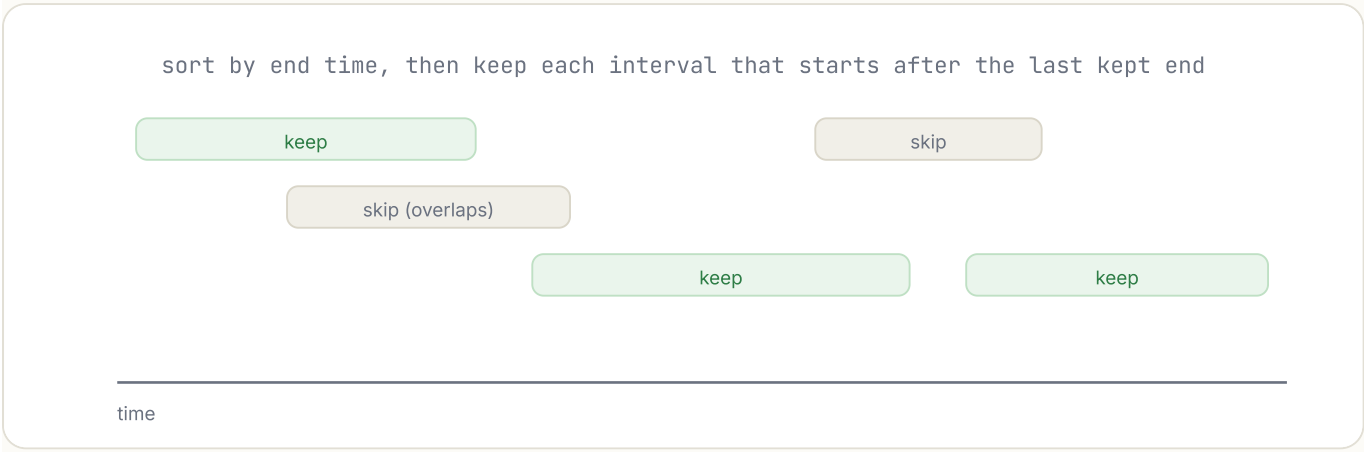
Do not assume greedy works just because it feels right. Coin change is the famous trap: greedily taking the biggest coin fails for some coin systems, which is exactly why that problem needs DP. Before trusting a greedy solution, try to break it with a small counterexample. If you cannot, and you can explain why the greedy choice never blocks a better future, you are probably safe.

DEEPER INTUITION

Why a local choice can be globally best

Greedy commits to the best looking choice at each step and never revisits it. The hard part is proving the local choice is safe, usually with an exchange argument: show that any optimal solution can be rewritten to agree with the greedy choice without getting worse. Sorting by the right key is the setup move. For scheduling the most non-overlapping intervals, finishing earliest leaves the

most room for the rest, so sorting by end time and greedily keeping whatever fits is provably optimal.



To pack the most non-overlapping intervals, sort by end time and greedily keep each one that fits.

Another worked example: Jump Game II

Reach the last index in the fewest jumps. Sweep left to right tracking the farthest index reachable. Each time you arrive at the boundary of the current jump, you are forced to jump, so increment the count and extend the boundary to the farthest seen. That greedy sweep gives the minimum number of jumps.

PYTHON

```python
def jump(nums):
    jumps = 0
    current_end = 0          # boundary reachable with the jumps so far
    farthest = 0
    for i in range(len(nums) - 1):
        farthest = max(farthest, i + nums[i])
        if i == current_end:      # must jump now
            jumps += 1
            current_end = farthest
    return jumps
```

JAVA

```java
int jump(int[] nums) {
    int jumps = 0, currentEnd = 0, farthest = 0;
    for (int i = 0; i < nums.length - 1; i++) {
        farthest = Math.max(farthest, i + nums[i]);
        if (i == currentEnd) {        // time to jump
            jumps++;
            currentEnd = farthest;
        }
    }
    return jumps;
}
```

GOING DEEPER

What kinds of problems this solves

USE THIS PATTERN FOR

Problems where taking the best looking choice at each step, usually after sorting by the right key, provably reaches the global best. Interval scheduling, reaching the end with jumps, and many buy or assign problems fit here.

Problem type	Classic examples
Interval scheduling	Non-overlapping Intervals, Minimum Number of Arrows to Burst Balloons, Merge Intervals
Reach and jumps	Jump Game, Jump Game II, Gas Station
Sequences and assignment	Best Time to Buy and Sell Stock II, Partition Labels, Assign Cookies, Task Scheduler
Building numbers or strings	Largest Number, Remove K Digits, Candy

The algorithm, in a bit more detail

A greedy algorithm commits to a locally optimal choice and never looks back. The hard part is not the code, it is being sure the local choice is safe. The usual proof is an exchange argument: show that any optimal solution can be rewritten to agree with the greedy choice without getting worse, so the greedy choice is never a mistake.

Sorting is the setup move for most greedy problems. Sort intervals by their end time and you can always keep the one that finishes earliest, leaving the most room for the rest. Choosing the correct sort key is where these problems are won or lost.

Variations you will run into

Greedy often pairs with a **sort**, and sometimes with a **heap** when you need the current best to change as you go, as in task scheduling. When a plain greedy fails, the problem usually needs dynamic programming instead, because a choice now affects options later in a way you cannot see locally.

Edge cases and gotchas

- Do not assume greedy works, confirm it with a proof or by testing small cases against brute force.
- The sort key is the crux, sort by end time for selection and by start time for merging.
- Greedy fails when a locally best choice blocks a better global outcome, that is your signal to switch to DP.

- Handle ties and empty inputs, which are common places greedy solutions quietly break.

STEP BY STEP

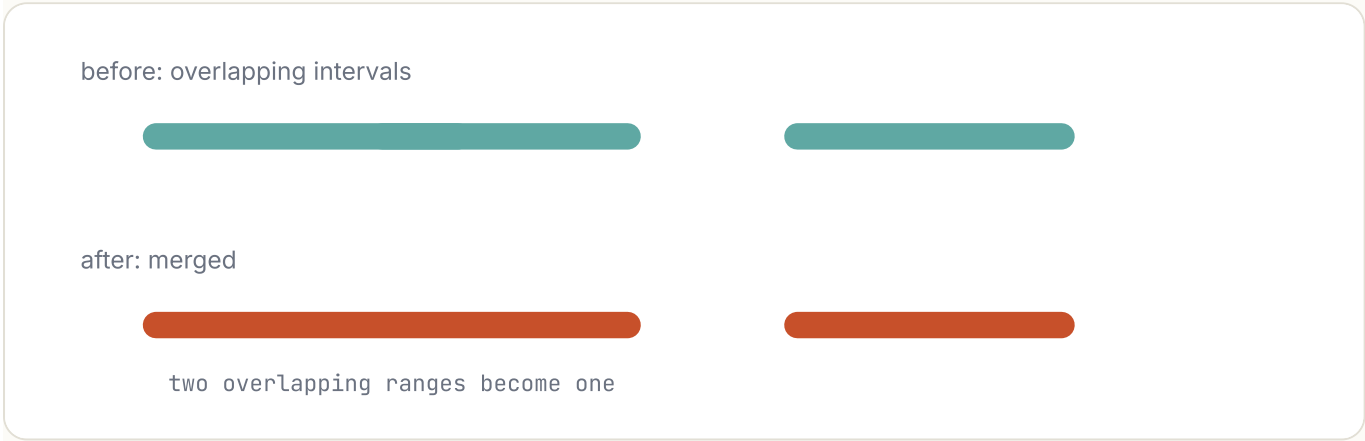
Jump Game on [2, 3, 1, 1, 4]. We track the farthest index we can reach and check that we never get stuck before it.

i	value	reachable so far	farthest now
0	2	0 ⩽ 0, ok	max(0, 0 + 2) = 2
1	3	1 ⩽ 2, ok	max(2, 1 + 3) = 4
2	1	2 ⩽ 4, ok	max(4, 2 + 1) = 4
3	1	3 ⩽ 4, ok	max(4, 3 + 1) = 4
4	4	4 ⩽ 4, reached the end	true

TOOLBOX

15 Intervals

An interval is just a start and an end, like a meeting from 2 to 4. Interval problems are about overlaps: merging them, counting how many overlap at once, or fitting them together. The universal first move is almost always the same: sort by start time.



Sort by start, then walk through and fuse anything that overlaps the current range.

REACH FOR INTERVAL TECHNIQUES WHEN YOU SEE

"Merge overlapping intervals", "insert a new interval", "can a person attend all meetings", "minimum meeting rooms needed", "maximum non-overlapping intervals". Ranges of time, ranges of numbers, or segments on a line are all intervals.

Worked example: merge intervals

Sort the intervals by start time. Walk through them one by one, keeping a current merged interval. If the next interval starts before the current one ends, they overlap, so stretch the current end to cover it. If it starts after, there is a gap, so push the current interval and start a new one.

PYTHON

```python
def merge(intervals):
    intervals.sort(key=lambda x: x[0])     # sort by start
    merged = [intervals[0]]
    for start, end in intervals[1:]:
        last = merged[-1]
        if start <= last[1]:               # overlaps the current interval
            last[1] = max(last[1], end)    # stretch the end
        else:
            merged.append([start, end])    # no overlap, new interval
    return merged
```

JAVA

```java
int[][] merge(int[][] intervals) {
    Arrays.sort(intervals, (a, b) -> a[0] - b[0]);   // sort by start
    List<int[]> merged = new ArrayList<>();
    merged.add(intervals[0]);
    for (int i = 1; i < intervals.length; i++) {
        int[] last = merged.get(merged.size() - 1);
        if (intervals[i][0] <= last[1]) {            // overlaps
            last[1] = Math.max(last[1], intervals[i][1]);
        } else {
            merged.add(intervals[i]);                // new interval
        }
    }
    return merged.toArray(new int[0][]);
}
```

Worked example: minimum meeting rooms

Given meeting time intervals, how many rooms do you need so none overlap in the same room? The clever version uses a min-heap of end times. Sort meetings by start. For each meeting, if the earliest-ending room is free by the time it starts, reuse that room; otherwise open a new room. The heap size at its peak is the answer.

PYTHON

```python
import heapq

def min_meeting_rooms(intervals):
    if not intervals:
        return 0
    intervals.sort(key=lambda x: x[0])     # by start time
    ends = []                              # min-heap of room end times
    for start, end in intervals:
        if ends and ends[0] <= start:      # a room freed up in time
            heapq.heapreplace(ends, end)   # reuse it
        else:
            heapq.heappush(ends, end)      # need a new room
    return len(ends)
```

JAVA

```java
int minMeetingRooms(int[][] intervals) {
    if (intervals.length == 0) return 0;
    Arrays.sort(intervals, (a, b) -> a[0] - b[0]);   // by start
    PriorityQueue<Integer> ends = new PriorityQueue<>();  // room end times
    for (int[] meeting : intervals) {
        if (!ends.isEmpty() && ends.peek() <= meeting[0]) {
            ends.poll();                 // reuse the freed room
        }
        ends.offer(meeting[1]);          // occupy a room until its end
    }
    return ends.size();
}
```

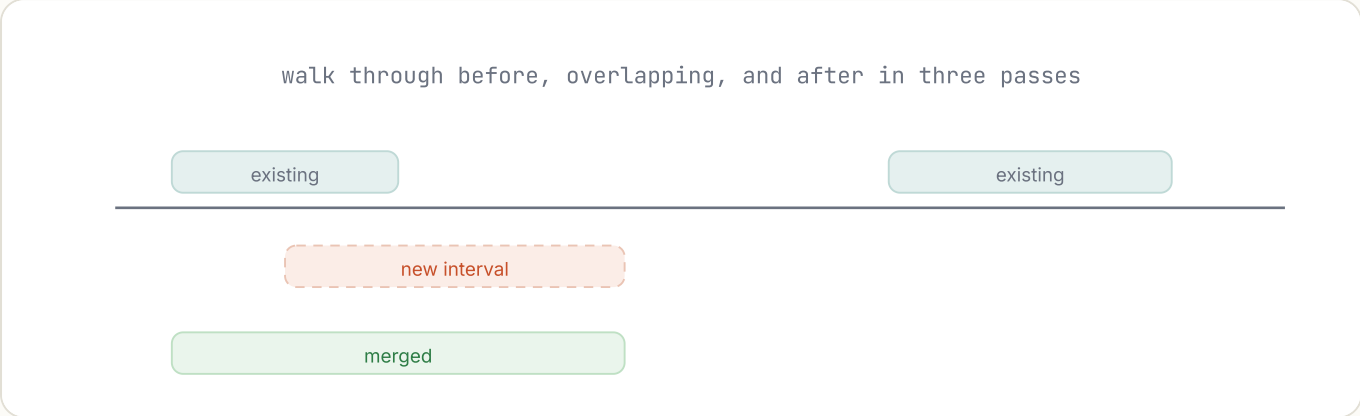
SORT BY THE RIGHT EDGE

Two intervals overlap when one starts before the other ends. For merging, sort by start. But for "maximum number of non-overlapping intervals" you sort by end time instead, then greedily keep each interval whose start is after the last kept end. Choosing the correct sort key is half the battle in interval problems.

DEEPER INTUITION

Why sorting first makes intervals easy

Almost every interval problem starts with a sort, because once intervals are in order you can sweep through them once and make a local decision at each step. Sort by start when you are merging, since overlaps become adjacent. Sort by end when you are selecting the most non-overlapping intervals. Choosing that sort key correctly is most of the battle, and the sweep that follows is short and mechanical.



Insert by handling the intervals entirely before the new one, then merging the overlapping ones, then the rest.

Another worked example: Insert Interval

Given sorted, non-overlapping intervals and a new one, insert it and merge. Handle three groups in order: intervals that end before the new one begins are copied as is, intervals that overlap are merged into the new one by widening its bounds, and the rest are copied after.

PYTHON

```python
def insert(intervals, new):
    result = []
    i, n = 0, len(intervals)
    while i < n and intervals[i][1] < new[0]:    # entirely before
        result.append(intervals[i]); i += 1
    while i < n and intervals[i][0] ⩽ new[1]:   # overlapping, merge
        new[0] = min(new[0], intervals[i][0])
        new[1] = max(new[1], intervals[i][1])
        i += 1
    result.append(new)
    while i < n:                                 # entirely after
        result.append(intervals[i]); i += 1
    return result
```

JAVA

```java
int[][] insert(int[][] intervals, int[] newInterval) {
    List<int[]> result = new ArrayList<>();
    int i = 0, n = intervals.length;
    while (i < n && intervals[i][1] < newInterval[0])    // before
        result.add(intervals[i++]);
    while (i < n && intervals[i][0] ⩽ newInterval[1]) { // overlap
        newInterval[0] = Math.min(newInterval[0], intervals[i][0]);
        newInterval[1] = Math.max(newInterval[1], intervals[i][1]);
        i++;
    }
    result.add(newInterval);
    while (i < n) result.add(intervals[i++]);            // after
    return result.toArray(new int[0][]);
}
```

GOING DEEPER

What kinds of problems this solves

USE THIS PATTERN FOR

Anything about ranges on a line: merging overlaps, inserting a new range, counting how many overlap at once, finding free time, or picking the most non-overlapping ranges. Sorting is almost always the first move.

Problem type	Classic examples
Merge and insert	Merge Intervals, Insert Interval, Interval List Intersections
Overlap counting	Meeting Rooms, Meeting Rooms II, My Calendar I, Employee Free Time
Selection and scheduling	Non-overlapping Intervals, Minimum Number of Arrows to Burst Balloons

The algorithm, in a bit more detail

Sort by start time, then sweep left to right keeping a current range. If the next interval starts before the current one ends, they overlap, so extend the current end. If it starts later, there is a gap, so close off the current range and open a new one. That single sweep merges everything in O(n log n), dominated by the sort.

For counting how many intervals are active at once, a min heap of end times works: for each meeting in start order, free any room whose end has already passed, then take a room. The largest the heap ever grows is the answer, which is how minimum meeting rooms is solved.

Variations you will run into

The sort key changes with the goal. **Sort by start** to merge. **Sort by end** to select the most non-overlapping intervals greedily. A **sweep line** that treats each start as plus one and each end as minus one turns overlap counting into a running sum.

Edge cases and gotchas

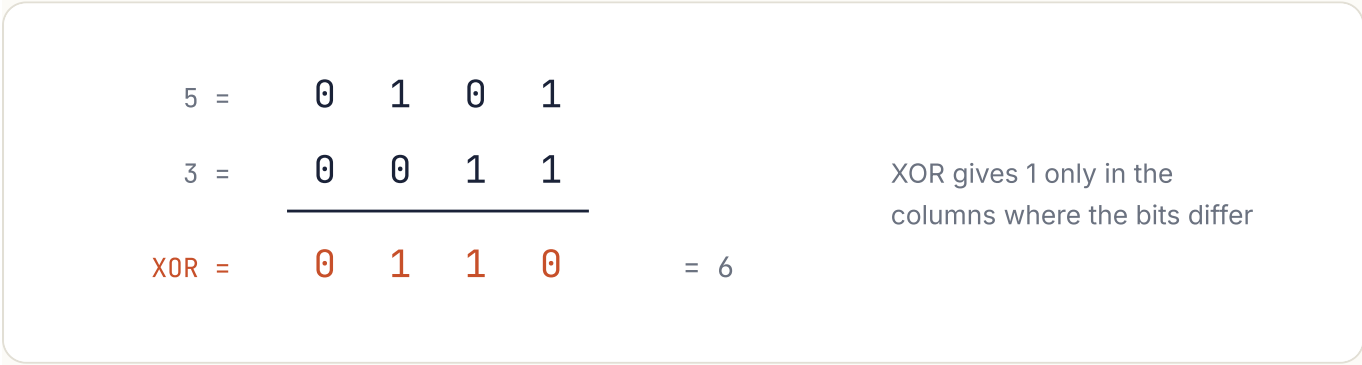
- Decide whether touching endpoints count as overlapping, since problems differ and it changes your comparison.
- Sort before you sweep, and pick start versus end based on whether you are merging or selecting.
- When inserting into an already sorted list, handle the intervals fully before, overlapping, and fully after separately.
- Watch empty input and a single interval, the usual boundary cases.

16 Bit manipulation

Computers store numbers as bits, a row of ones and zeros. A few problems become almost free if you work directly with those bits. You do not need to love binary, you just need a handful of moves.

The operators, in plain terms

Operator	What it does
`&` AND	1 only where both bits are 1. Used to test or clear bits.
`|` OR	1 where either bit is 1. Used to set bits.
`^` XOR	1 where the bits differ. A number XORed with itself is 0.
`~` NOT	Flips every bit.
`<<` left shift	Shifts bits left, which multiplies by 2 each step.
`>>` right shift	Shifts bits right, which divides by 2 each step.



XOR compares bit by bit and keeps only the differences.

REACH FOR BITS WHEN YOU SEE

"Find the one number that appears once while all others appear twice" (XOR). "Count the set bits." "Is this a power of two." "Generate all subsets using a bitmask." Anything about flags, toggles, or on/off states packed into a single integer.

Worked example: the single number

Every number appears twice except one. Find the lonely one. XOR is perfect here: any number XORed with itself cancels to 0, and XOR with 0 leaves a number unchanged. So XOR the whole array together; the pairs cancel out and the single number survives. This is O(n) time and O(1) space, beating a hash set.

PYTHON

```python
def single_number(nums):
    result = 0
    for x in nums:
        result ^= x        # pairs cancel, the lonely number remains
    return result
```

JAVA

```java
int singleNumber(int[] nums) {
    int result = 0;
    for (int x : nums) {
        result ^= x;       // pairs cancel out
    }
    return result;
}
```

Handy one-line tricks worth memorizing

Goal	Trick
Is n even?	`n & 1` is 0
Is n a power of two?	`n > 0 and (n & (n - 1)) == 0`
Turn off the lowest set bit	`n & (n - 1)`
Multiply / divide by 2	`n << 1` and `n >> 1`
Check if bit i is set	`(n >> i) & 1`

Worked example: count the set bits

Count how many 1s are in the binary form of a number. The elegant way uses the trick above: `n & (n - 1)` removes the lowest 1 each time. So loop, removing one 1 per step, and count the steps. It runs only as many times as there are set bits, not the full bit width.

PYTHON

```python
def count_bits(n):
    count = 0
    while n:
        n &= n - 1        # clears the lowest set bit
        count += 1
    return count
```

JAVA

```java
int countBits(int n) {
    int count = 0;
    while (n != 0) {
        n &= (n - 1);     // clear the lowest set bit
        count++;
    }
    return count;
}
```

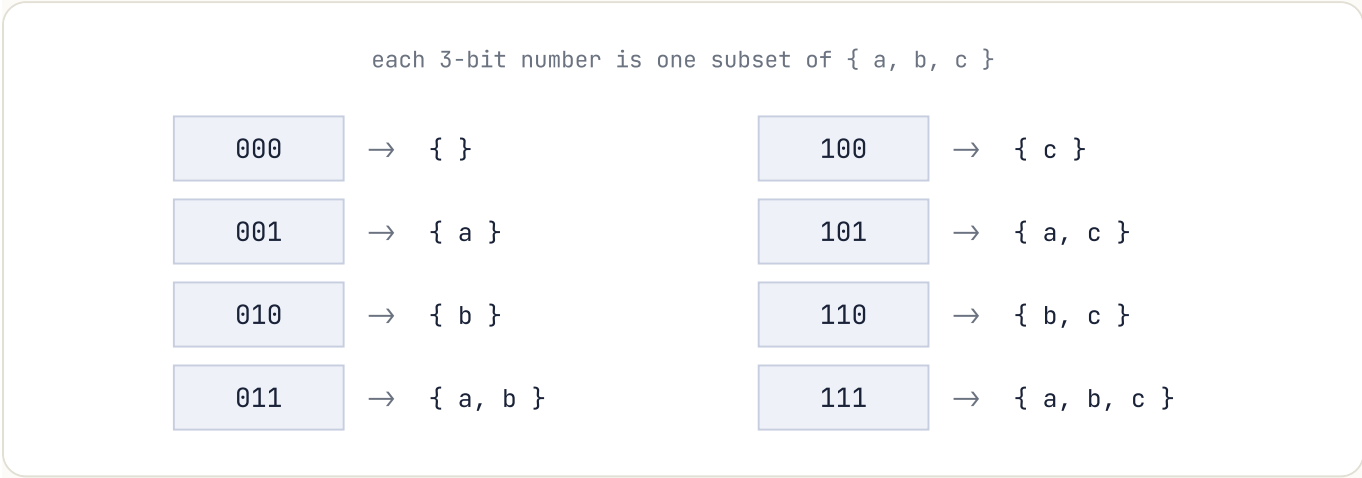
WATCH OUT

Java integers are 32-bit and signed, so right-shifting a negative number keeps the sign bit; use the unsigned shift `>>>` when you want zeros pulled in from the left. Python integers have unlimited size and no fixed width, so some bit tricks that rely on overflow behave differently. When a problem mixes negatives and bit shifts, test with a negative input on purpose.

DEEPER INTUITION

Why bits are secretly sets

The three moves that carry most bit problems are simple: XOR cancels pairs since a number XORed with itself is zero, the expression n and n minus one clears the lowest set bit, and a mask lets you test, set, or clear any single bit. The deeper idea is that an integer is a tiny set: each bit says whether an element is in or out, so counting from zero up to two to the n walks through every possible subset. That is the backbone of bitmask dynamic programming for small inputs.



Treating an integer's bits as in-or-out flags, the numbers 0 to 2^n minus 1 enumerate every subset.

Another worked example: Missing Number

An array holds the numbers 0 to n with exactly one missing. XOR every index together with every value, plus n itself. Every number that is present cancels with its matching index, and the one missing number is left standing. O(n) time and O(1) space.

PYTHON

```python
def missing_number(nums):
    result = len(nums)              # start with n
    for i, x in enumerate(nums):
        result ^= i ^ x             # XOR all indices and values
    return result                   # the missing one survives
```

JAVA

```java
int missingNumber(int[] nums) {
    int result = nums.length;
    for (int i = 0; i < nums.length; i++)
        result ^= i ^ nums[i];      // pairs cancel, missing remains
    return result;
}
```

GOING DEEPER

What kinds of problems this solves

USE THIS PATTERN FOR

Problems about toggles and flags, finding the odd one out among pairs, enumerating subsets with a bitmask, counting set bits, and quick checks like power of two. Also bitmask dynamic programming for small sets.

Problem type	Classic examples
XOR tricks	Single Number, Single Number II, Single Number III, Missing Number, Find the Difference
Counting and checks	Number of 1 Bits, Counting Bits, Power of Two, Reverse Bits
Bitmask enumeration and DP	Subsets via bitmask, Sum of Two Integers, Maximum XOR of Two Numbers, Partition to K Equal Sum Subsets

The algorithm, in a bit more detail

The three moves that carry this chapter are simple. XOR cancels pairs, since a number XORed with itself is zero, so XORing a whole array leaves only the element that appears an odd number of times. The expression n and n minus one clears the lowest set bit, which lets you count set bits in as many steps as there are ones. And a mask lets you test, set, or clear any single bit with a shift.

The more advanced use is treating an integer as a set. Each bit says whether an element is in or out, so a number from zero up to two to the n covers every subset. That is the backbone of bitmask dynamic programming for problems with small n, like assigning tasks or the travelling salesman on a handful of cities.

Variations you will run into

You will use **plain XOR** for odd one out problems, **n and n minus one** for counting and power of two checks, and **bitmask as a set** for enumerating subsets or as a DP state. Shifts double or halve a number cheaply.

Edge cases and gotchas

- In Java, right shifting a negative number keeps the sign bit, so use the unsigned shift with three angle brackets when you want zeros pulled in.
- Python integers have no fixed width, so tricks that rely on overflow behave differently than in Java or C.
- Bitwise operators have low precedence, so wrap them in parentheses inside comparisons.
- Be careful mixing signed values and shifts, and test with a negative input on purpose.

17 Pattern cheat sheet

This is the page to reread before a contest or an interview. It compresses the whole book into signals, tools, and typical costs. When a problem lands in front of you, run down this list and something will click.

Signal to pattern

The problem says or implies	Try this pattern	Typical time
Seen this value before, count, or pair up	Hash map or set	`O(n)`
Sorted array, pair summing to a target, palindrome	Two pointers	`O(n)`
Longest or shortest contiguous run	Sliding window	`O(n)`
Matching brackets, next greater, undo	Stack	`O(n)`
Sorted data, or "smallest value that works"	Binary search	`O(log n)`
Reverse, middle, cycle in a chain	Linked list pointers	`O(n)`
Depth, height, path sum in a tree	DFS recursion	`O(n)`
Level by level, shortest unweighted path	BFS with a queue	`O(n)`
Kth largest, top K, running median	Heap	`O(n log k)`
All subsets, permutations, combinations	Backtracking	exponential
Islands, connected groups, reachability	DFS or BFS on a graph	`O(nodes + edges)`
Prerequisites, build order	Topological sort	`O(nodes + edges)`
Min or max cost, number of ways, overlapping subproblems	Dynamic programming	`O(states)`
Locally best choice provably safe	Greedy	`O(n log n)` with a sort
Overlapping ranges, meeting rooms	Intervals, sort first	`O(n log n)`
Appears once among pairs, flags, toggles	Bit manipulation	`O(n)`

The decision flow, in words

Start by reading the constraints. If n is tiny (up to about 20), an exponential backtracking solution is probably fine and maybe expected. If n is large (100,000 or more), you need roughly O(n) or O(n log n), which rules out nested loops.

Next, look at the shape of the data. Is it sorted, or would sorting help? That points to binary search, two pointers, or a greedy sweep. Is it a tree or a grid or a network? That points to DFS and BFS. Is it a string or array asking for a contiguous best? That is a sliding window.

Finally, look at what is being asked. "How many ways" or "min / max with subproblems" is dynamic programming. "All of something" is backtracking. "The Kth best" is a heap. "Have I seen this" is a hash map.

> **WHEN YOU ARE TOTALLY STUCK**
>
> Write the brute force anyway. Getting a slow but correct solution on the board does three things: it proves you understood the problem, it often reveals the repeated work you can cut, and it gives you partial credit. Then ask the one question that unlocks most optimizations: "what am I recomputing, and can I remember it instead?"

Complexity reference for common operations

Operation	Cost
Hash map or set lookup, insert, delete	`O(1)` average
Array index access	`O(1)`
Append to a dynamic array or list	`O(1)` amortized
Insert or delete in the middle of an array	`O(n)`
Sorting	`O(n log n)`
Binary search on a sorted array	`O(log n)`
Heap push or pop	`O(log n)`
BFS or DFS over a graph	`O(nodes + edges)`

18 A study plan that works

Talent is overrated here. Consistency and the right practice beat raw brains. Here is a plan that turns the patterns in this book into instinct over a few weeks, without burning you out.

The weekly rhythm

Pick one pattern per few days rather than jumping around randomly. Depth beats variety early on. For each pattern, read its chapter here, then solve four to six problems that use it, from easy to medium. Do not move to the next pattern until you can rewrite the core template from memory.

How to practice one problem

1. **Try it cold for about twenty minutes.** Real struggle is where the learning happens. Do not peek early.
2. **If stuck, read only a small hint,** not the full solution. Often "this is a sliding window" is enough to unstick you.
3. **After solving or reading the answer, close everything and rewrite it from scratch.** If you cannot, you have not learned it yet, so repeat.
4. **Write one sentence** about the trick that made it work. This is your personal cheat sheet, and it will be worth more than any book at review time.

> A SENSIBLE ORDER TO LEARN THE PATTERNS
>
> Arrays and hashing, then two pointers and sliding window, then stack and binary search. Next linked lists, then trees and BFS or DFS, then heaps. Then the hard hitters: backtracking, graphs, and dynamic programming. Finish with greedy, intervals, and bits. Each stage builds on the ones before, so resist skipping ahead.

Spaced repetition beats cramming

A problem you solved today should be revisited in a few days, then a week later, then a couple of weeks after that. Each time it should feel a little easier. The ones that still feel hard on the second

pass are exactly the ones worth your time. Keep a short list of "solved but felt shaky" problems and cycle back to them.

SIMULATE THE REAL THING

Before an interview or contest, do a few problems under a timer, talking out loud as if someone is listening. Explaining your thinking while you code is a separate skill from just coding, and it is the one most people neglect. Practicing it removes most of the panic on the real day.

THE TRAPS THAT STALL PEOPLE

Solving hundreds of problems without ever revisiting them, so nothing sticks. Reading solutions before really trying, which feels productive but teaches almost nothing. Chasing hard problems too early instead of building a solid base of easy and medium ones. And skipping the "rewrite from memory" step, which is the single highest-value habit in this whole plan.

One last thing

Everyone who is good at this was once bad at it and felt lost staring at problems that now seem easy. The gap between "I have no idea" and "oh, this is just a sliding window" is not intelligence, it is exposure. Keep showing up, keep rewriting from memory, and keep noting the trick. The patterns in this book are finite, and once they are in your hands, most of LeetCode turns into recognition. You have got this.

PART TWO

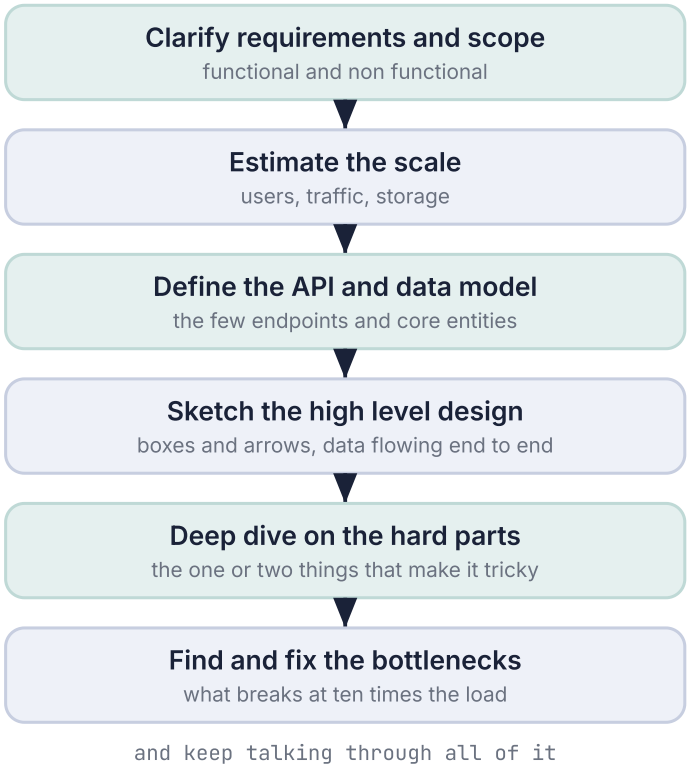
System Design

How large systems are actually built, from load balancers and caches to databases, replication, and message queues. We start with the fundamentals and the trade offs, then design real products end to end: a URL shortener, WhatsApp, Instagram, Amazon, Amazon S3, YouTube, and Uber.

FUNDAMENTALS

19 Approaching system design

System design questions feel open ended and a little scary, but they follow a script just like coding problems do. Nobody expects one perfect answer. The interviewer wants to watch how you think, how you weigh trade offs, and how you drive the conversation. Learn the script and the fear mostly goes away.



A repeatable script for almost any design question, top to bottom.

The six moves, in order

Start by **clarifying requirements**. Split them into functional (what the system must do, like send a message or shorten a link) and non functional (how it must behave, like handle millions of users,

stay up, respond in under a hundred milliseconds). Ask whether it is read heavy or write heavy, how many users, and whether strong consistency matters. These answers shape every later decision.

Next, **estimate the scale** with rough math so your design is grounded in real numbers rather than vibes. Then **define the API and data model**, just the handful of endpoints and the core entities, because they force you to be concrete. Now **draw the high level design**, boxes for the client, load balancer, services, cache, database, and queue, and make data flow from a request all the way to a response.

With the skeleton in place, **deep dive on the hard parts**, whatever makes this system special, like unique ID generation for a URL shortener or fanout for a news feed. Finally, **hunt for bottlenecks** and show how you scale past them by adding caches, replicas, shards, or queues.

FUNCTIONAL VS NON FUNCTIONAL, KNOW THE DIFFERENCE

Functional requirements are features: post a tweet, follow a user, search. Non functional requirements are qualities: availability, latency, consistency, durability, and cost. Interviewers care most about how you handle the non functional side, because that is where the interesting trade offs live.

DRIVE THE CONVERSATION

Say your assumptions out loud and write them down. State the numbers you are using. When you pick a database or a cache, say why, and name the trade off you are accepting. A design interview is a discussion, not a monologue, so a running commentary of your reasoning is exactly what earns the score.

20 Back of the envelope estimation

Estimation is how you prove a design can actually hold the load. You are not after exact figures, you are after the right order of magnitude, enough to decide whether one database is fine or you need a hundred. A few numbers and a little multiplication carry you a long way.

Latency numbers worth memorizing

These are approximate, but the relative gaps are the point. Memory is fast, disk is slow, and crossing the planet is very slow.

Operation	Rough time
Read from main memory (RAM)	about 100 nanoseconds
Read 1 MB sequentially from memory	tens of microseconds
Round trip inside one data center	about 0.5 milliseconds
Read from a solid state drive (SSD)	about 100 microseconds
Read from a spinning disk (seek)	about 10 milliseconds
Round trip across continents	about 100 to 150 milliseconds

The storage and traffic math

Powers of ten keep it simple: a thousand is 10^3 (kilo), a million is 10^6 (mega), a billion is 10^9 (giga), a trillion is 10^12 (tera). A day has 86400 seconds, which you can round to 10^5 for quick division.

To size traffic, take daily active users times the actions each one does per day, then divide by the seconds in a day to get requests per second. Then multiply by two or three for peak, since traffic is bursty, not flat.

A WORKED ESTIMATE

Suppose a service has 100 million daily active users, and each makes 10 requests a day. That is 1 billion requests a day. Divide by roughly 100000 seconds and you get about 10000 requests per second on average, so plan for around 25000 to 30000 at peak. If each request reads 1 KB, that is tens of megabytes per second of bandwidth. Now you know whether a single server is a joke or a plan.

ROUND AGGRESSIVELY

Use 10^5 for seconds in a day, treat a KB as 1000 bytes, and round user counts to the nearest power of ten. The interviewer is checking that you can reason about scale, not that you can do long division under pressure.

BUILDING BLOCKS

21 Scaling and load balancing

There are only two ways to handle more load. Make one machine bigger, or add more machines. The first hits a ceiling fast and gives you a single point of failure. The second is how real systems grow, and a load balancer is the piece that makes it possible.

Vertical vs horizontal scaling

Vertical scaling means a beefier server: more CPU, more memory. It is simple and needs no code changes, but you cannot buy an infinitely large machine, and if it dies, everything dies with it. Horizontal scaling means many ordinary servers working together. It scales almost without limit and survives individual failures, but it forces you to design for a fleet: no server can hold state that another server would need.

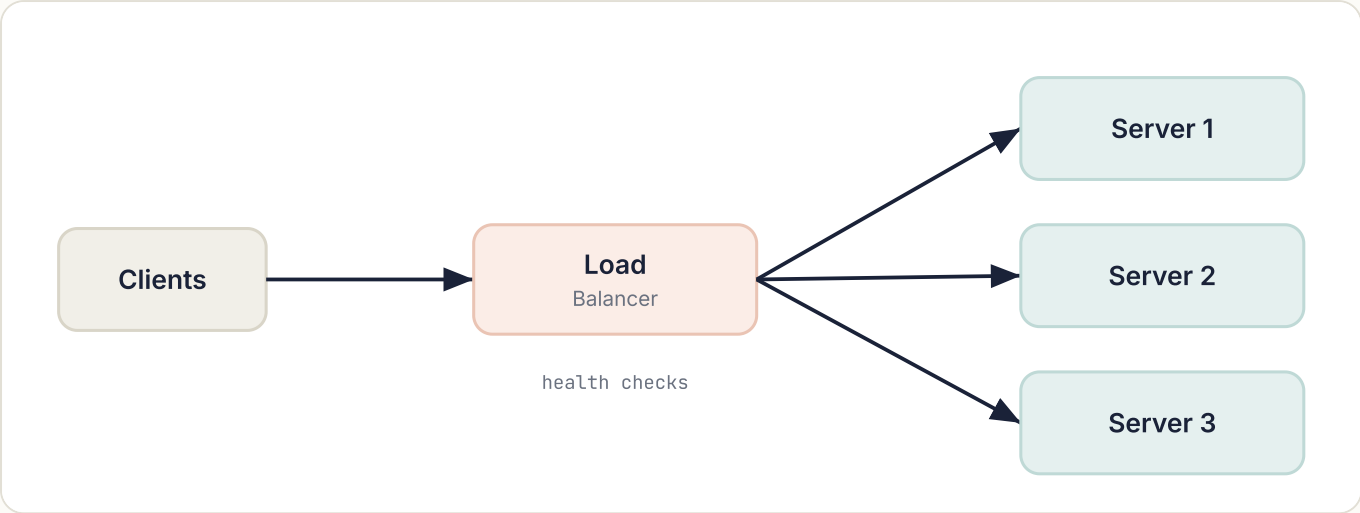
PROS

- Horizontal scaling has no hard ceiling, just add machines
- One machine failing does not take the system down
- You can scale up for peak and down to save money

CONS

- Your services must be stateless, which takes design effort
- You now need a load balancer and shared storage for state
- Distributed systems bring their own bugs and complexity

The load balancer



One entry point spreads traffic across many identical, stateless servers.

A load balancer sits in front of your servers and spreads incoming requests across them, so no single machine is overwhelmed. It also checks the health of each server and quietly stops sending traffic to one that stops responding. Common ways to choose a server are round robin (take turns), least connections (send to the least busy), and hashing on something like the client IP (so a client tends to hit the same server).

Balancers work at two levels. A layer 4 balancer routes on IP and port without looking inside the request, which is fast. A layer 7 balancer reads the actual request, so it can route by URL path or cookie, at a little more cost. The key rule that makes all of this work is to keep your servers **stateless**: store session data in a shared cache or database, so any server can handle any request and you can add or remove servers freely.

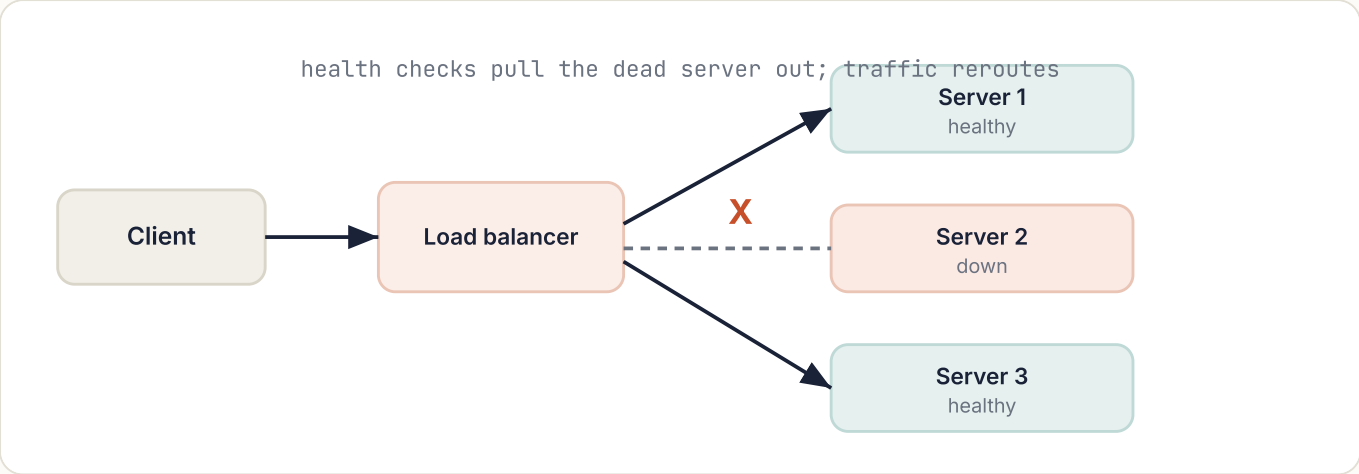
THE HIDDEN SINGLE POINT OF FAILURE

If you have one load balancer, it becomes the thing that can take everything down. In real systems the balancer itself is made redundant, with a standby ready to take over, so no single box is the weak link.

GOING DEEPER

Why stateless servers are the whole trick

Horizontal scaling only works because every server is interchangeable, and that only holds if servers keep no local state. Push session data into a shared store so any server can handle any request, which is what lets you add or remove machines freely and lets the load balancer send a user to a different server each time. The balancer also runs continuous health checks, so the moment a server stops responding it is pulled from rotation and traffic reroutes to the healthy ones, making a single failure invisible. The balancer itself is made redundant with a standby, so it does not become the one thing that can take everything down.



When a server fails its health check, the balancer stops sending it traffic and the rest carry on.

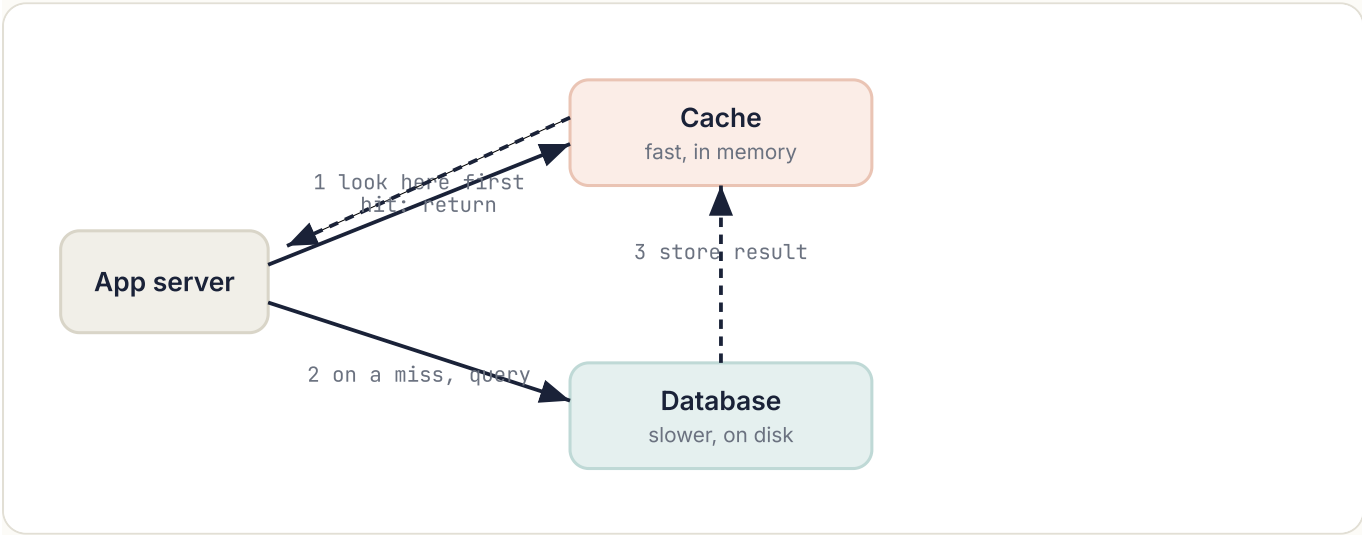
Balancer algorithm	How it picks a server
Round robin	take turns evenly, simple and common
Least connections	send to the least busy server
IP hash	same client tends to hit the same server
Weighted	bigger servers get proportionally more traffic

22 Caching and CDNs

A cache is a small, fast copy of data kept close to where it is needed, so you avoid a slow trip to the database every time. Caching is the single highest leverage move for read heavy systems, which is most systems. It also brings the hardest problem in the field: knowing when the cached copy is stale.

Where caching happens

Caching lives at many layers. The browser caches on the client. A content delivery network caches near the user at the edge. Your application caches hot data in memory using something like Redis or Memcached. Even the database caches its own frequent queries. Each layer you add removes load from the layer behind it.



Cache aside: check the cache, fall back to the database, then fill the cache.

Read and write strategies

The most common read pattern is **cache aside**: the app checks the cache, and on a miss it reads the database and then stores the value in the cache for next time. For writes you choose how the cache and database stay in step. **Write through** updates both on every write, so the cache is always fresh but writes are a little slower. **Write back** updates the cache immediately and the database later, which is fast but risks losing data if the cache dies first. **Write around** writes only to the database and lets the cache fill on the next read, which avoids caching data nobody reads.

Eviction and the staleness problem

A cache is small, so it must throw things out. The usual policy is least recently used, which evicts whatever has not been touched in the longest time. Others include least frequently used and a simple time to live that expires entries after a set period.

The famous hard part is **invalidation**. When the underlying data changes, the cached copy is now wrong, and serving stale data can be anything from harmless to dangerous. The tools are a time to live so entries expire on their own, explicit invalidation when you know data changed, and versioned keys so a new version simply misses the old cache.

PROS

- Reads get dramatically faster and cheaper
- The database is shielded from repetitive load
- A CDN cuts latency by serving content near the user

CONS

- Stale data is possible, and invalidation is genuinely hard
- A cache adds a moving part that can fail or fill up
- A sudden loss of the cache can stampede the database

CONTENT DELIVERY NETWORKS

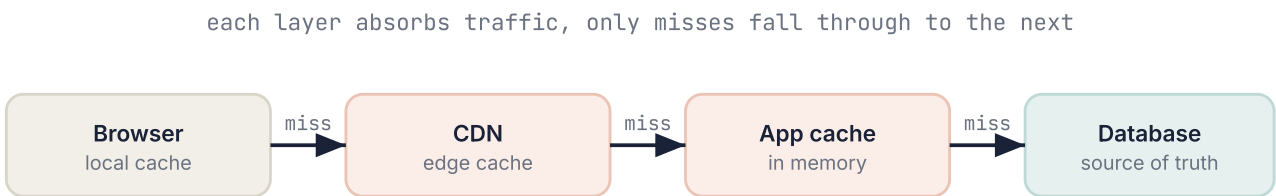
A CDN is a global cache for static content like images, video, scripts, and styles. It stores copies at edge locations around the world, so a user in Tokyo is served from a nearby machine instead of your origin server across the ocean. This slashes latency and takes a huge load off your own servers.

GOING DEEPER

Layers of cache, and the stampede to avoid

Caching is rarely one cache. It is a stack: the browser, then a CDN at the edge, then an in memory cache in front of the database, and each layer that hits shields the layers behind it. The classic failure mode is a cache stampede: a popular key expires and thousands of requests miss at the same instant, all hammering the database together and sometimes knocking it over. The fixes are to let only one request refill a missing key while the others briefly wait, to stagger expiry times so keys

do not all die at once, and to serve slightly stale data while a fresh copy is fetched in the background.



Caching is a stack of layers. A hit at any layer returns early, so only misses reach the database.

THREE WAYS TO KEEP A CACHE FRESH

Time to live expires entries on their own after a set period, which is simple but can serve stale data until it lapses. Explicit invalidation deletes or updates the entry the moment the source changes, which is precise but needs you to catch every change. Versioned keys sidestep the problem by writing to a new key on each change, so a stale entry is simply never read again.

STEP BY STEP

How one read flows through cache aside, the most common caching pattern.

1. The application receives a read request for some data.
2. It looks in the cache first.
3. On a hit, it returns the cached value immediately, never touching the database.
4. On a miss, it reads the value from the database.
5. It writes that value into the cache, then returns it, so the next identical read is a fast hit.

23 Databases: SQL vs NoSQL

Picking a database is one of the biggest decisions in a design, and the honest answer is always "it depends on your access patterns." The split people talk about is relational (SQL) versus everything else (NoSQL), so it helps to know what each is genuinely good and bad at.

Relational databases

A relational database stores data in tables with a fixed schema, and it can join tables together to answer complex queries. Its superpower is **ACID transactions**, which guarantee that a group of changes either all happen or none do, keeping data correct even under concurrency. This is why banking, orders, and inventory almost always sit on a relational store like PostgreSQL or MySQL.

PROS

- Strong consistency and reliable transactions
- Flexible, powerful queries with joins
- A schema keeps data structured and validated
- Mature, well understood, everywhere

CONS

- Harder to scale writes across many machines
- A rigid schema slows down rapid change
- Joins get expensive at very large scale

NoSQL databases

NoSQL is a family, not one thing. Key value stores like Redis and DynamoDB are simple and blazing fast. Document stores like MongoDB hold flexible JSON like records. Wide column stores like Cassandra handle massive write volume across many machines. Graph databases like Neo4j specialize in relationships. What they share is a flexible schema and an easier path to horizontal scale, often by relaxing strict consistency to eventual consistency.

PROS

- Scales horizontally across many machines with ease

CONS

- Weaker consistency, often only eventual
- Limited or no joins, so you design around access patterns

- Flexible schema fits changing or messy data
- Very high throughput for the right access pattern
- Often simpler and cheaper at huge scale

- Transactions are limited or absent in many systems
- Easy to model data in a way you later regret

Indexing, the quiet workhorse

An index is a separate sorted structure, usually a B tree, that lets the database find rows without scanning the whole table. It turns a slow full scan into a fast lookup. The trade off is that every index costs extra storage and slows down writes, because each insert or update must also update the index. So you index the columns you filter and sort on, and no more.

CHOOSE BY ACCESS PATTERN

Ask how the data will be read and written before you pick. Complex queries and strict correctness point to relational. Enormous write volume with simple lookups points to a wide column or key value store. Do not choose based on which name sounds modern, choose based on what your reads and writes actually look like.

GOING DEEPER

Indexes, and normalize versus denormalize

An index is usually a B tree, a shallow and very wide tree that lets the database descend to any row in a few hops instead of scanning the whole table. That speed is not free: every index costs storage and slows writes, since each insert must update it too, so you index only the columns you filter and sort on. The other big lever is the shape of the data. A relational design normalizes, splitting data into tidy tables that are joined at query time, which avoids duplication but makes reads do work. A document design denormalizes, embedding related data together, so a read is a single fetch but an update may have to touch many copies. You are trading write simplicity for read speed.

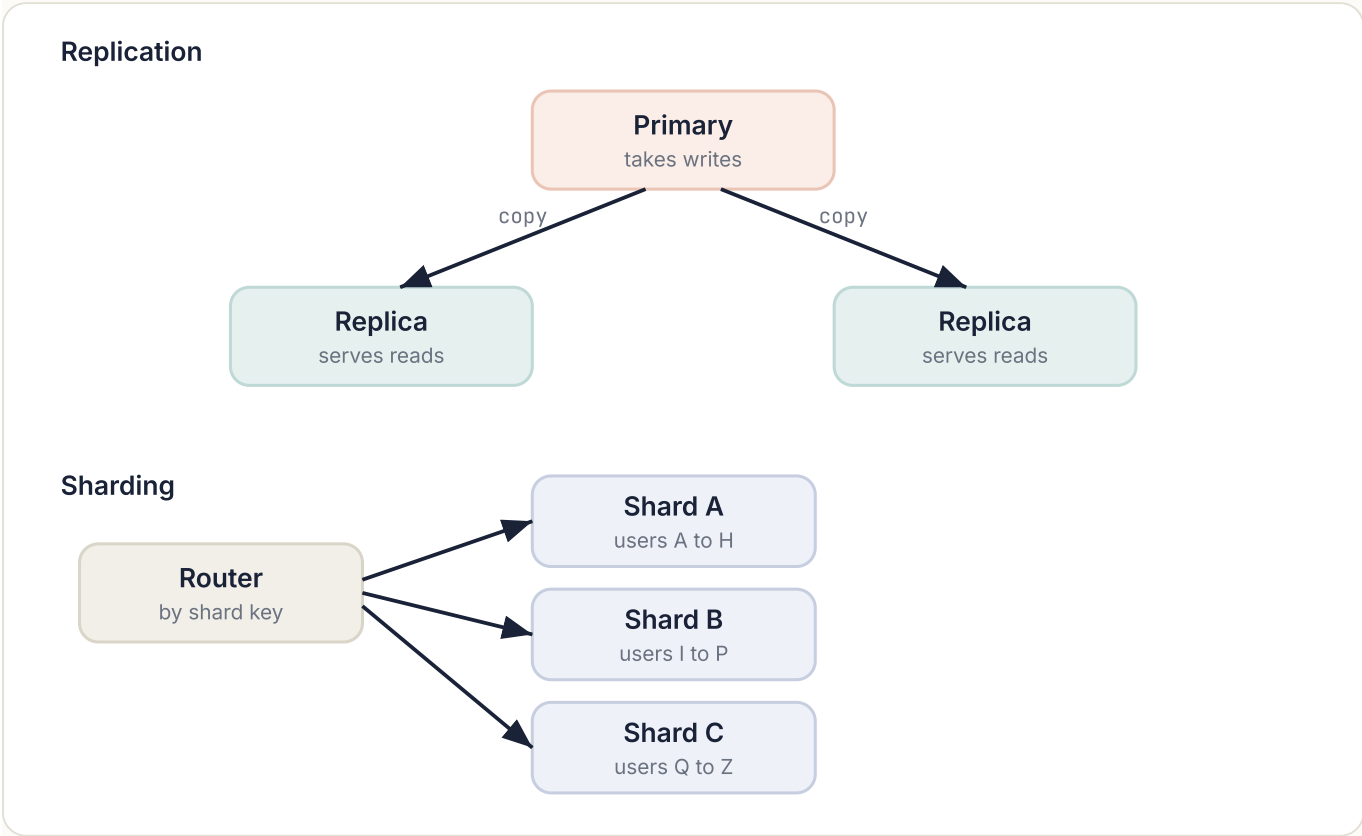
Normalizing splits data into tidy tables joined on read. Denormalizing embeds it together for a single fast read.

ACID VERSUS BASE

Relational systems aim for ACID: transactions are atomic, consistent, isolated, and durable, so data is always correct even under heavy concurrency. Many distributed NoSQL systems instead offer BASE: basically available, soft state, eventually consistent, trading immediate correctness for availability and scale. Neither is better in the abstract, they match different needs, strict money movement versus a high volume feed.

24 Replication and sharding

One database eventually runs out of room, either for traffic or for data. Two techniques fix the two different problems. Replication makes copies for safety and read speed. Sharding splits the data so writes and storage spread across many machines. They solve different things and are often used together.



Replication copies the same data for safety and read scale. Sharding splits different data for write scale.

Replication

Replication keeps copies of the same data on several machines. The common setup is one **primary** that takes all the writes and several **replicas** that copy from it and serve reads. This gives you two wins: if the primary dies, a replica can take over, and read heavy traffic spreads across many replicas.

The catch is how the copy happens. **Asynchronous** replication is fast because the primary does not wait for replicas, but a crash can lose the most recent writes that had not copied yet. **Synchronous**

replication waits for replicas to confirm, so nothing is lost, but every write is slower. Most systems pick a spot on that spectrum based on how much they fear losing a few seconds of data.

PROS

- Survives a machine failure with a replica ready to take over
- Read traffic scales across many replicas
- Replicas can sit closer to users in other regions

CONS

- Writes still funnel through a single primary
- Replicas can lag, so reads may be slightly stale
- Failover and promoting a new primary is fiddly

Sharding, also called partitioning

Sharding splits the data itself so different machines hold different rows. You pick a **shard key**, like user id, and route each record to a shard based on it. Range based sharding groups by ranges of the key, which is simple but can create hotspots. Hash based sharding spreads keys evenly by hashing them. **Consistent hashing** is the refinement that lets you add or remove a machine while moving only a small slice of the data instead of reshuffling everything.

PROS

- Write throughput and storage scale nearly without limit
- Each shard is smaller, so its own operations stay fast
- Consistent hashing keeps rebalancing cheap

CONS

- Queries that span shards are slow and awkward
- A bad shard key creates hotspots that ruin the balance
- Transactions across shards are hard or impossible
- More machines means more operational overhead

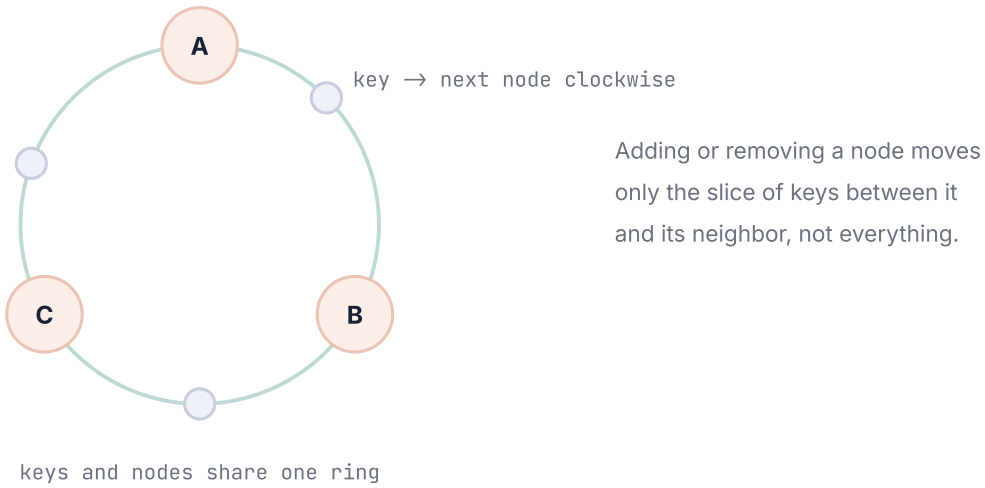
THE SHARD KEY IS EVERYTHING

Choose a shard key that spreads load evenly and matches how you query. If most requests hit one popular key, that shard becomes a hotspot and you are back to a single overloaded machine. Picking this well is one of the highest stakes decisions in a sharded design.

GOING DEEPER

Consistent hashing, and choosing a new leader

Plain hashing across N machines has a nasty flaw: change N and almost every key moves to a different machine. Consistent hashing fixes this by placing both nodes and keys on a ring, where each key belongs to the next node clockwise. Add or remove a node and only the keys between it and its neighbor move, a small slice instead of the whole dataset. For replication, when the primary dies the replicas run a leader election to promote a new primary and clients are redirected to it. The subtle part is doing this without two nodes both believing they are the primary, which would corrupt data.



Consistent hashing places nodes and keys on a ring, so changing the node count reshuffles only a small slice.

PROS

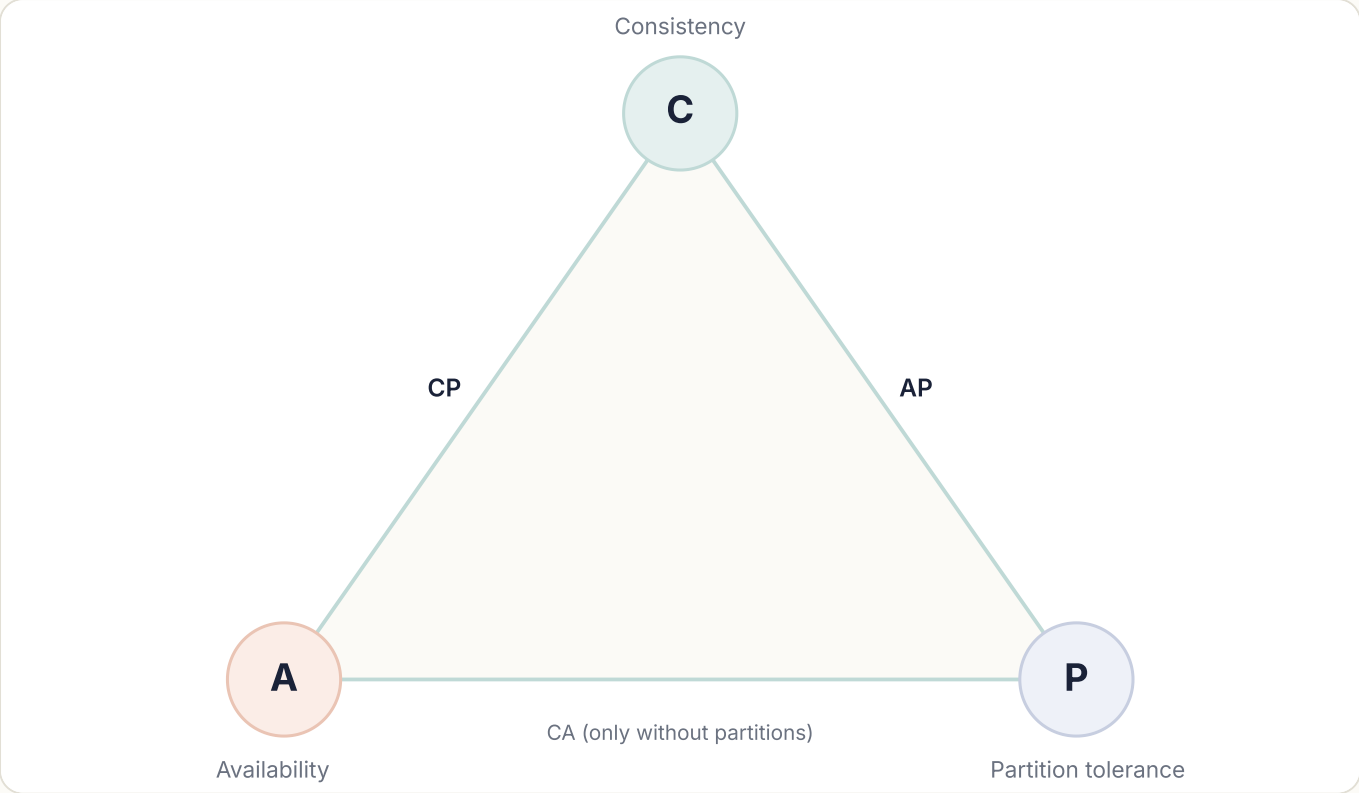
- Async replication is fast, since the primary does not wait for replicas
- Reads scale across many replicas placed near users

CONS

- Async can lose the last few writes if the primary crashes
- Sync replication is safe but makes every write slower

25 CAP theorem and consistency

Once data lives on more than one machine, physics forces a trade off. The CAP theorem names it. In the moment two parts of your system cannot talk to each other, you have to choose between giving every request a correct answer and giving every request an answer at all.



During a network partition you must give up either strong consistency or availability.

What the three letters mean

Consistency here means every read sees the most recent write, so all machines agree. **Availability** means every request gets a response, even if some machines are down. **Partition tolerance** means the system keeps working when the network between machines drops messages, which in the real world it eventually will.

The theorem says you can only fully guarantee two of the three at once. Since network partitions are a fact of life in any distributed system, partition tolerance is not really optional. So the real choice is **CP versus AP**: when a partition happens, do you refuse to answer rather than risk a wrong answer (CP), or do you keep answering and accept that some replicas are temporarily out of date (AP)?

How this shows up

A banking ledger leans **CP**: it would rather reject a request than show two people the same money. A social feed or a shopping cart often leans **AP**: showing a slightly stale like count or reconciling a cart later is fine, and staying available matters more.

CONSISTENCY IS A SPECTRUM

Beyond the strict either or, there are useful middle grounds. **Strong consistency** means reads always reflect the latest write. **Eventual consistency** means replicas converge given a little time. **Read your writes** guarantees you at least see your own recent changes. Real systems pick different levels for different features, tight for payments, loose for view counts.

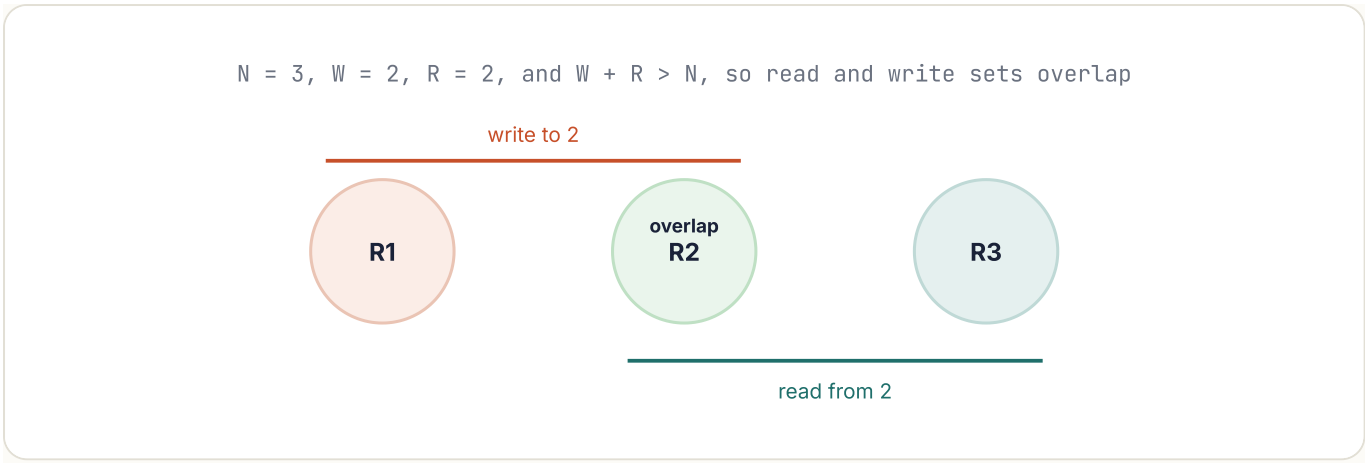
PACELC, THE HONEST FOOTNOTE

CAP only talks about behavior during a partition. PACELC adds the rest: even when everything is healthy (the Else case), you still trade a little latency for a little consistency. Waiting for more replicas to confirm a write is safer but slower. That quiet trade off is always present, not just during failures.

GOING DEEPER

Tuning consistency with quorums

You do not have to choose strong or eventual for the whole system. Quorums let you dial it per operation. With N replicas, if every write must reach W of them and every read must reach R of them, then as long as W plus R is greater than N the read set and the write set always share at least one replica, so a read is guaranteed to see the latest write. Turn W and R up for stronger consistency at the cost of slower, less available operations, or down for speed and weaker guarantees. This is exactly how systems like Dynamo make the trade off a knob rather than a fixed choice.



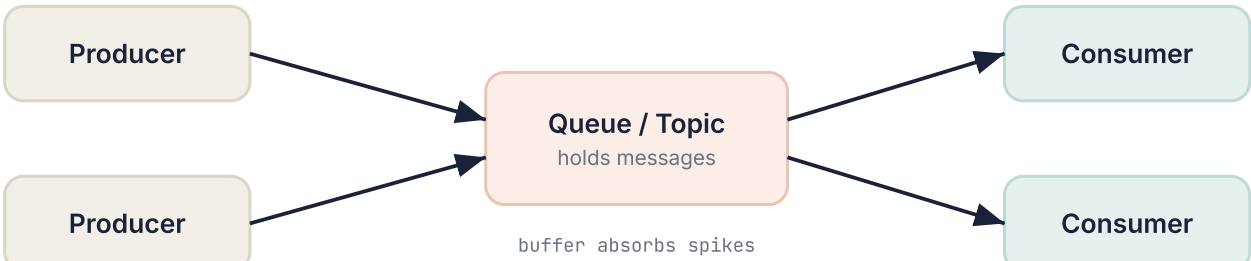
With enough overlap between the write set and the read set, a read is guaranteed to see the latest write.

THE CONSISTENCY SPECTRUM

Strong consistency means every read reflects the latest write. Eventual consistency means replicas converge given a little time. Read your writes guarantees you at least see your own recent changes. Causal consistency preserves cause and effect ordering. Real systems mix these, tight for payments, loose for view counts, rather than picking one globally.

26 Message queues and async processing

Not everything has to happen while the user waits. Sending an email, resizing an image, updating a feed, these can happen in the background. A message queue is the piece that lets one part of the system hand off work to another without waiting, which makes systems more resilient and much easier to scale.



A queue decouples producers from consumers, so each side works at its own pace.

Why go asynchronous

A queue sits between a **producer** that creates work and a **consumer** that does it. The producer drops a message and moves on immediately, so the user is not stuck waiting. If a traffic spike arrives, the queue buffers the flood and the consumers drain it at a steady pace, so nothing falls over. If a consumer crashes, the message waits safely until another consumer picks it up.

Queues versus publish and subscribe

A plain **queue** is point to point: each message is handled by exactly one consumer, which is perfect for a pool of workers sharing a task list. **Publish and subscribe** broadcasts each message on a topic to every interested subscriber, which is perfect when one event needs to trigger many independent reactions. Log based systems like Kafka blur the line by keeping an ordered, replayable log that many consumer groups can read at their own position.

PROS

- The user gets a fast response while work happens later
- Spikes are absorbed instead of crashing the system
- Producers and consumers scale and fail independently
- Failed work can be retried automatically

CONS

- More moving parts and a new system to operate
- Results are eventual, not immediate, which can confuse users
- Exactly once delivery is very hard, so you design for duplicates
- Debugging a flow spread across queues is harder

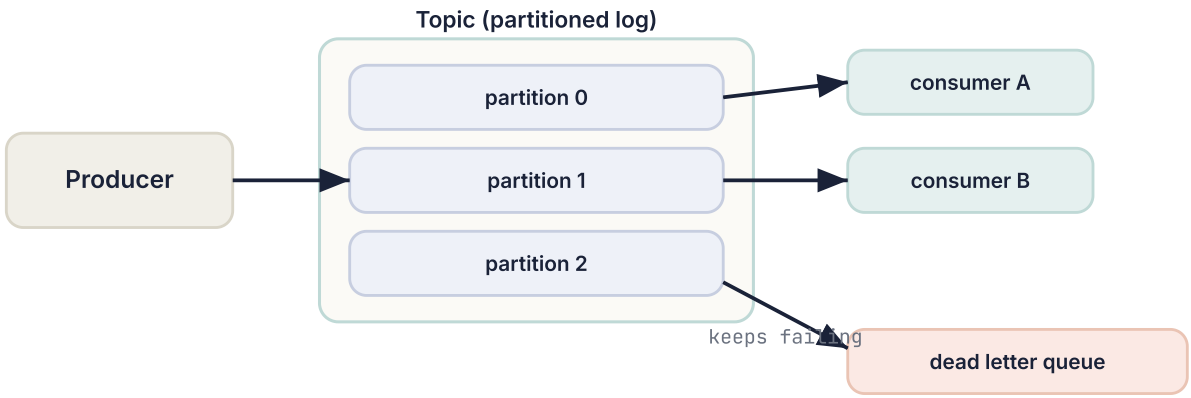
AT LEAST ONCE PLUS IDEMPOTENCY

Guaranteeing a message is delivered exactly once is famously hard. The practical pattern is at least once delivery, which may deliver a message twice, combined with idempotent consumers that produce the same result no matter how many times they process the same message. A dead letter queue catches messages that keep failing so they do not block everything else.

GOING DEEPER

Partitions, ordering, and messages that will not process

A log based system like Kafka splits a topic into partitions, and order is only guaranteed within a partition, so you choose a partition key such as user id to keep related messages in order. Consumers form a group and divide the partitions between them, so adding consumers raises throughput. Because delivery is usually at least once, a message can arrive twice, so consumers must be idempotent, producing the same result no matter how many times they see it. A message that keeps failing is moved to a dead letter queue after a few retries, so one poison message does not block everything behind it.



A topic is split into ordered partitions. Consumers share them, and messages that keep failing move aside to a dead letter queue.

AT LEAST ONCE PLUS IDEMPOTENCY

Guaranteeing exactly once delivery is famously hard, so the practical recipe is at least once delivery combined with idempotent consumers. Give each message a stable id and have the consumer ignore ids it has already processed. That way a duplicate is harmless, and you get effectively once behavior without the cost of true exactly once machinery.

27 More building blocks

A few more components show up in almost every design. You do not need to design each from scratch, but you should know what they are for and when to reach for them.

API gateway

An API gateway is a single front door for all client requests. It handles the cross cutting concerns you do not want scattered across every service: authentication, rate limiting, request routing, and sometimes response shaping. Clients talk to one endpoint, and the gateway forwards to the right service behind it.

Rate limiting

Rate limiting protects a system from abuse and overload by capping how many requests a client can make in a window. The **token bucket** is the classic approach: a bucket refills tokens at a steady rate, each request spends one, and when the bucket is empty requests are rejected or delayed. It allows short bursts while enforcing a long term average. The **leaky bucket** and **sliding window** are common alternatives with slightly different burst behavior.

Object storage

Large files like images, video, and backups do not belong in a relational database. Object storage, the S3 style service, is built for exactly this: cheap, durable, effectively unlimited storage for big blobs, usually served to users through a CDN.

Search

Searching text with a database `LIKE` query does not scale. A search engine like Elasticsearch builds an **inverted index**, a map from each word to the documents that contain it, so full text queries stay fast even over huge datasets. You keep it in sync with your main database as data changes.

Observability

Once a system is distributed, you cannot debug it by guessing. The three pillars are **logs** (what happened), **metrics** (numbers over time like latency and error rate), and **tracing** (following one request across many services). Alerting on the right metrics is what turns a 3 a.m. outage into a 3 a.m. page before users notice.

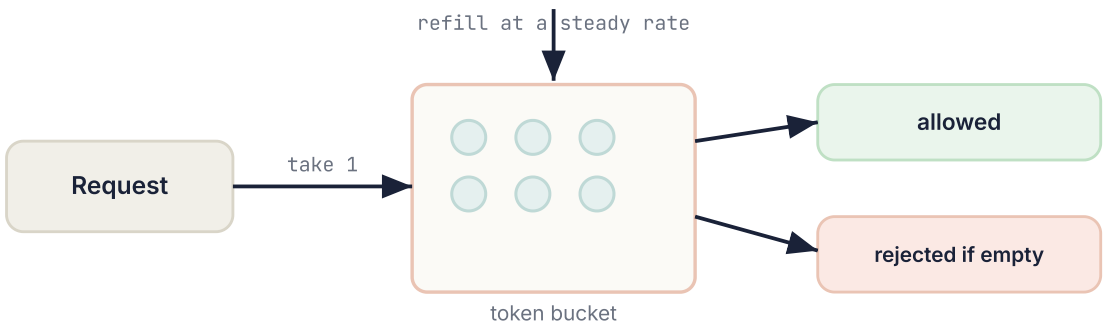
YOU RARELY BUILD THESE YOURSELF

In a design discussion, naming the right off the shelf building block is the goal, not reinventing it. Saying "put an API gateway here for auth and rate limiting, object storage for the media, and a search index kept in sync with the database" shows exactly the judgment interviewers are looking for.

GOING DEEPER

Rate limiting up close, with the token bucket

The token bucket is the workhorse of rate limiting. A bucket holds up to a fixed number of tokens and refills at a steady rate. Each request removes a token, and if the bucket is empty the request is rejected or delayed. This cleverly allows short bursts, up to the bucket size, while still enforcing a long term average equal to the refill rate. A sliding window counter is the other common choice, smoothing the hard edges of a fixed window so a burst straddling the boundary cannot sneak through double the limit. You place this at the API gateway so every service behind it is protected in one spot.



A bucket refills tokens at a steady rate. Each request spends one, and an empty bucket means rejection.

FIXED WINDOW VERSUS SLIDING WINDOW

A fixed window counts requests per clock interval, which is simple but lets a client fire a full quota at the end of one window and another at the start of the next, briefly doubling the rate. A sliding window weights the previous window as it moves, smoothing that spike out. The token bucket avoids the issue differently, by capping the burst to the bucket size.

DESIGN CASE STUDIES

28 Design walkthrough: a URL shortener

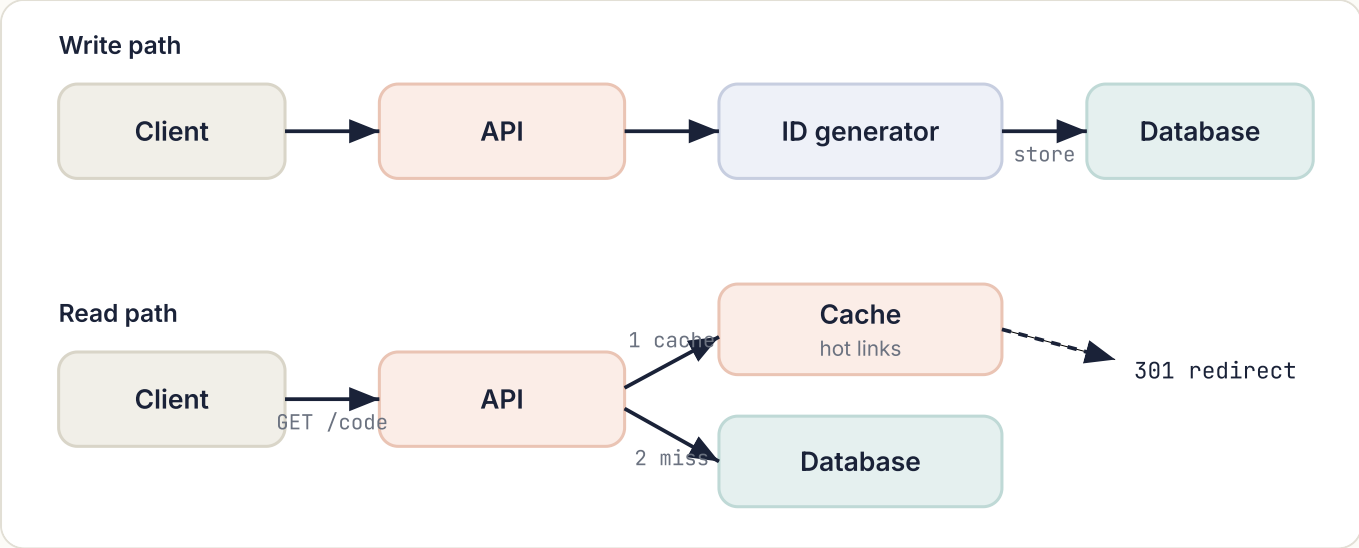
This is the classic warm up design. It looks tiny, but it touches unique ID generation, heavy read caching, and horizontal scale, so it is a great place to practice the whole framework end to end.

Requirements and scale

Functionally, take a long URL and return a short code, and when someone visits the short code, redirect them to the original. Non functionally, it is extremely read heavy, since a link is created once but clicked many times, and redirects must be fast. That read to write ratio drives the whole design toward caching and read replicas.

The API and data model

Two endpoints cover it: a write, POST with the long URL that returns a short code, and a read, GET on the short code that returns a redirect. The data is just a mapping: short code to long URL, plus a creation time. It is a simple key value shape, which means it shards and caches beautifully.



Writes generate a unique code and store it. Reads hit the cache first, then redirect.

The interesting part: generating the code

You need a short, unique code for each URL. Three approaches come up. **Hashing** the URL and taking the first few characters is simple but produces collisions you must handle. **Encoding an auto incrementing id** in base 62 (letters and digits) gives short, guaranteed unique codes, but a single counter is a bottleneck and reveals how many links exist. A **key generation service** pre generates random unique codes in advance and hands them out, which avoids both collisions and a hot counter.

PROS

- Base 62 counter gives the shortest possible unique codes
- A key generation service avoids collisions and a hot counter
- The key value shape shards and caches extremely well

CONS

- A plain hash risks collisions you must detect and retry
- A single global counter is a scaling and secrecy problem
- Custom or vanity codes add a uniqueness check on write

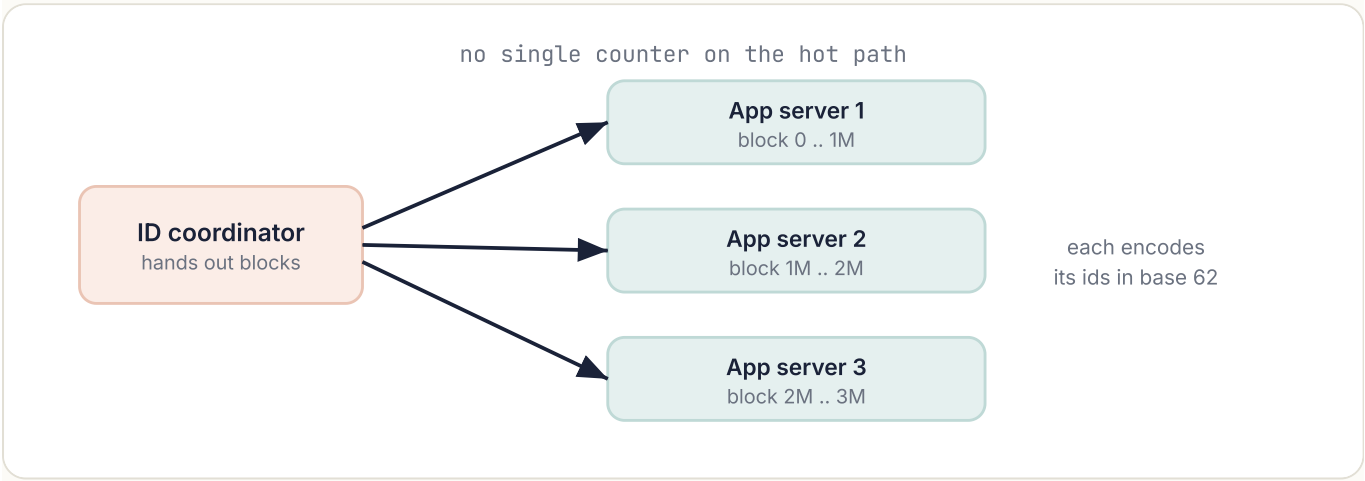
Making it scale

Because reads dominate, cache the popular codes in memory so most redirects never touch the database. Put the mapping in a store that shards by code, and add read replicas so lookups spread out. For the redirect itself, a 301 (permanent) lets browsers cache it and cut future load, while a 302 (temporary) keeps every click coming to you, which is useful if you want to count clicks.

GOING DEEPER

Generating unique codes without a bottleneck

The tempting design, a single auto incrementing counter, becomes a hot spot that every write must pass through, and it quietly leaks how many links exist. The fix is to hand out ranges. A coordinator gives each app server a block of ids, say a million at a time, and each server then encodes ids from its own block into base 62 with no coordination per request. An alternative is a key generation service that pre generates random unique codes in advance and hands them out, which also removes collisions. Base 62 keeps the codes short by using digits together with upper and lower case letters.



Each server gets a block of ids up front and encodes locally, so no request waits on a shared counter.

301 VERSUS 302, AND COUNTING CLICKS

A 301 permanent redirect lets browsers cache the mapping, so future clicks skip your server entirely, which is fastest but means you cannot count those clicks. A 302 temporary redirect sends every click back through you, which is what you want if analytics on each visit matter. The choice is a straight trade between speed and measurement.

STEP BY STEP

The two paths through a URL shortener, one for creating a link and one for using it.

Creating a short link

1. The client sends the long URL to the API.
2. The service generates a unique short code, for example by encoding a counter in base 62.
3. It stores the mapping from code to long URL in the database.
4. It returns the short code, so the full short link is ready to share.

Using a short link

1. A visitor requests the short code.
2. The service checks the cache first, since popular links are read constantly.
3. On a cache miss it looks up the code in the database and fills the cache.
4. It responds with a redirect to the original long URL, and the browser follows it.

29 Design walkthrough: WhatsApp

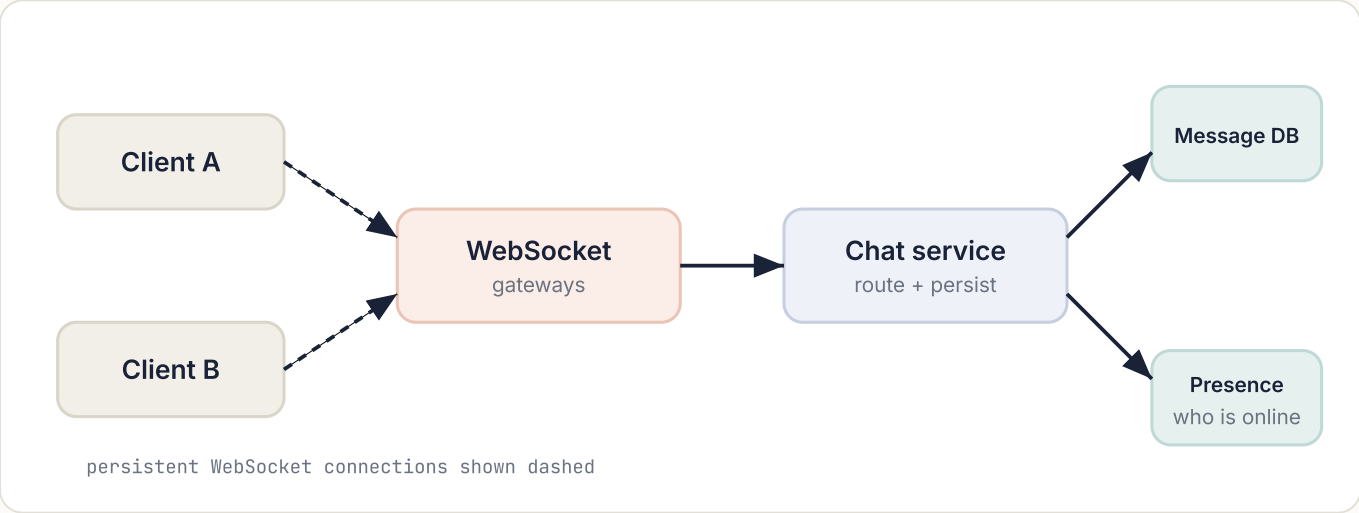
WhatsApp is the classic chat design. It forces you to think about real time delivery, which most systems never need, plus the twist of holding hundreds of millions of live connections open at once and getting a message from one phone to another in milliseconds.

Requirements

Support one to one and group messages, deliver them in real time, show who is online, keep message history, and ideally show delivery and read receipts. The non functional headline is low latency delivery and a huge number of simultaneous open connections.

The real time transport

A normal request and response cannot push a message to you when it arrives. The answer is a **WebSocket**, a connection that stays open in both directions, so the server can push a message the instant it is received. Clients hold a live WebSocket to a gateway server, and that persistent connection is the backbone of the whole system.



Clients hold a live connection to a gateway, which routes messages through the chat service.

How a message flows

When Client A sends a message, it travels over the open connection to a gateway, then to the chat service, which persists it and figures out where the recipient is connected. Because the recipient may be attached to a different gateway, the system keeps a mapping of user to gateway, often in a presence store, so it can route the message to the right server and push it down the recipient's connection. Messages are stored keyed by conversation with a sequence number, so history loads in order and nothing arrives scrambled.

PROS

- WebSockets push messages instantly with low overhead
- Stateless gateways scale horizontally for more connections
- A queue lets you deliver to offline users and send push alerts

CONS

- Holding millions of open connections is resource heavy
- Routing to the right gateway needs a presence mapping
- Ordering and exactly once delivery need real care
- Group chat fanout multiplies the work per message

OFFLINE DELIVERY AND GROUPS

If a recipient is offline, drop the message on a queue and send a push notification, then deliver history when they reconnect. For group chat, the message fans out to every member, so a queue plus workers handles the multiplication without blocking the sender. Per conversation sequence numbers keep everyone's view consistent.

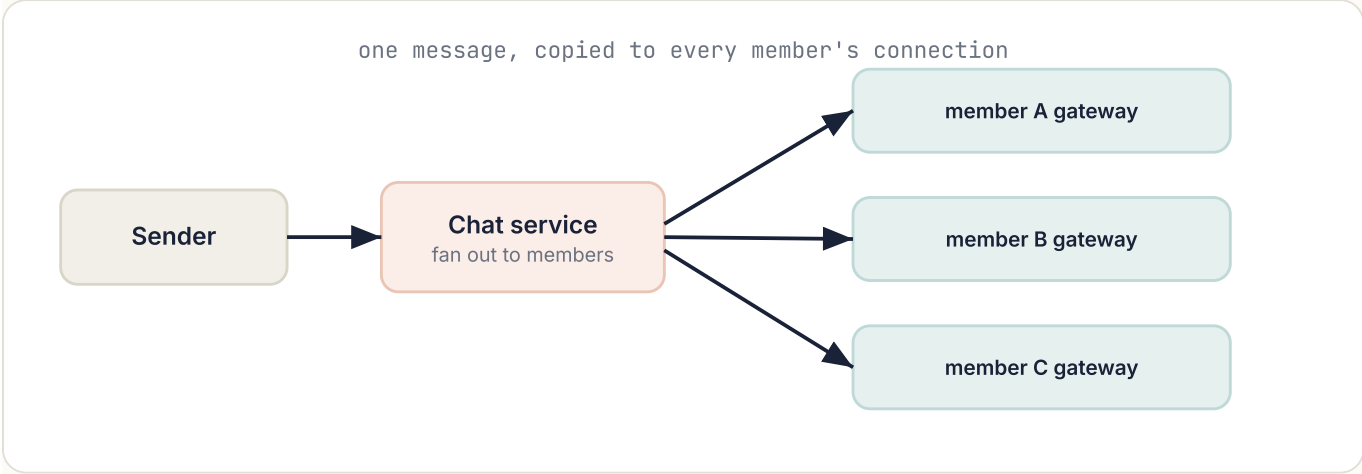
WHAT MAKES WHATSAPP SPECIFIC

A few product choices shape the whole design. Identity is your phone number, so there is no username system to build. Messages are end to end encrypted, so the servers route ciphertext they cannot read and keep as little as possible once a message is delivered. Each message shows one tick when it reaches the server, two when it reaches the other device, and blue when it is read, which are just delivery acknowledgements flowing back over the same connection. Presence signals like last seen and typing are lightweight pushes. Because delivery is usually instant, the server can often drop a message from storage as soon as the recipient confirms it, which keeps the footprint small even at enormous scale.

GOING DEEPER

Group messages and keeping order

One to one delivery is simple, but a group message has to reach every member, and each of them may be connected to a different gateway server. The chat service looks up where each member is connected and fans the message out to all of them, using a queue so a large group does not block the sender while it delivers. Ordering matters too, so each conversation stamps its messages with an increasing sequence number, and clients display by that sequence. That keeps everyone's view of the chat consistent even when messages happen to arrive out of order.



A group message fans out to every member, each possibly on a different gateway, using a queue to avoid blocking the sender.

MULTIPLE DEVICES AND OFFLINE DELIVERY

A user may be on a phone and a laptop at once, so a message is delivered to every active device and marked read across them together. If a device is offline, the message waits on a per user queue and a push notification is sent, then the history syncs when that device reconnects, so nothing is missed.

STEP BY STEP

How a single WhatsApp message travels from one phone to another.

1 The sender's phone sends the message over its open WebSocket connection to a gateway.

2 The gateway passes it to the chat service, which persists it and looks up where the recipient is connected.

3 If the recipient is online, the service pushes the message straight down their gateway connection.

4 The recipient's app acknowledges receipt, which flows back to the sender as the second tick.

5 If the recipient is offline, the message waits on a queue and a push notification is sent, then it is delivered when they reconnect.

30 Design walkthrough: Instagram

Instagram is a feed sitting on top of a media problem. The feed is the timeline of posts from people you follow, and it carries the same hard trade off every feed does: do you do the work when someone posts, or when someone reads? On top of that, the posts are photos and videos, which changes how you store and deliver them.

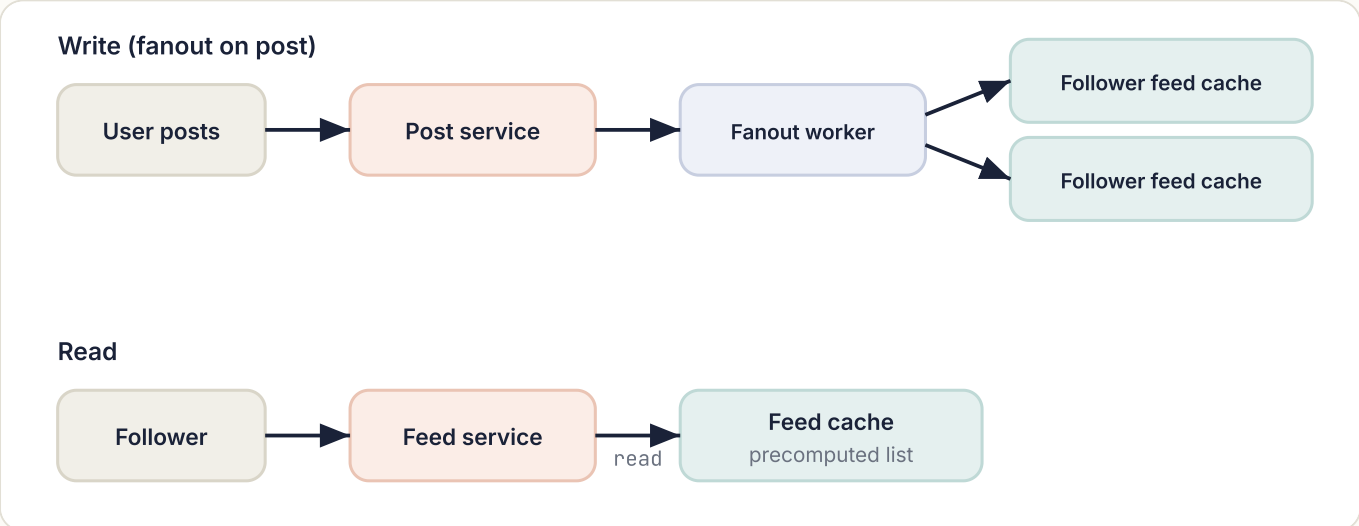
Requirements

Users follow others and see a feed of recent posts, usually newest first or ranked by relevance. Reads must be fast and are far more frequent than writes. The tricky part is that some users have millions of followers, which breaks the simple approaches.

The core choice: push versus pull

Fanout on write, the push model, does the work when you post: the post is immediately written into the precomputed feed of every follower. Reads then become a trivial lookup, which is wonderful, but a single post by someone with millions of followers triggers millions of writes.

Fanout on read, the pull model, does the work when you open the app: the feed is built on the fly by gathering recent posts from everyone you follow. Writes are cheap, but reads become expensive and slow, especially if you follow many people.



Push writes a post into every follower's feed ahead of time, so reads are a simple lookup.

PROS

- Push makes reads a fast, precomputed lookup
- Pull keeps writes cheap and handles celebrities well
- A hybrid gets the best of both for different user types

CONS

- Push explodes on users with millions of followers
- Pull makes every feed load do heavy gathering work
- Keeping precomputed feeds fresh adds complexity
- Ranking on top of either approach adds more cost

The hybrid that real systems use

The practical answer is a mix. Use **push** for ordinary users, so their followers get fast precomputed feeds. Use **pull** for the handful of celebrities with enormous followings, whose posts are merged in at read time instead of fanned out to millions. This hybrid sidesteps the worst case of each while keeping the common case fast.

THE CELEBRITY PROBLEM DRIVES THE DESIGN

If every account were small, pure push would be perfect. The entire reason for the hybrid is that a few accounts have millions of followers, and fanning a single post out to all of them would be crippling. Naming this trade off is exactly what a strong answer to this question sounds like.

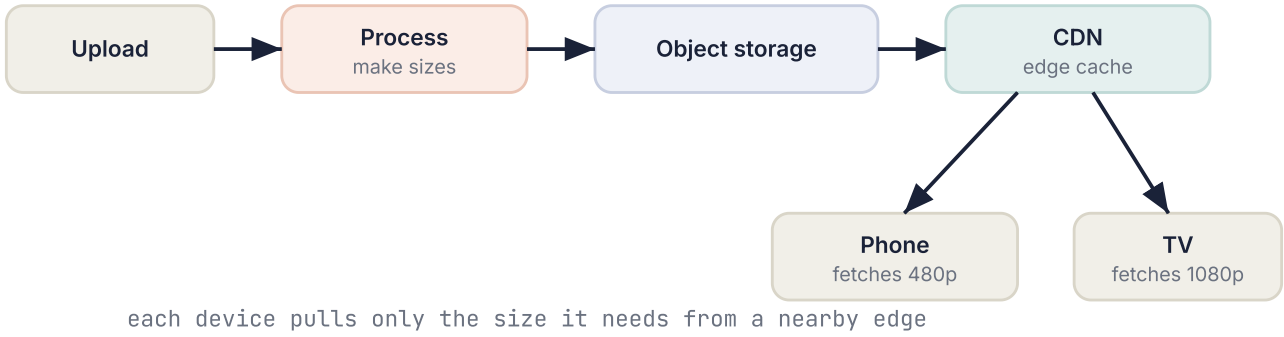
WHAT MAKES INSTAGRAM SPECIFIC

The feed logic is the push and pull mix from this chapter, but the media is the real work. Photos and videos are large, so they never live in the main database. On upload they go to object storage, get processed into several resolutions, and are served through a CDN close to the viewer, while the database holds only small records like the media id, caption, and author. Stories and the explore page are separate ranked surfaces layered on the same building blocks. The heavy lifting is storing and delivering media fast, not storing text.

GOING DEEPER

The media pipeline behind a single photo

When a photo is uploaded it does not go into a database, it goes to object storage and kicks off processing that produces several sizes and formats, a thumbnail, a feed sized version, and a full size original. Those variants live in object storage and are served through a CDN, so each device pulls just the resolution it will actually display from a nearby edge. The database only stores a small record pointing at the media. This split is why feeds load quickly around the world: the heavy bytes are cached close to users, and the app never downloads more pixels than it shows.



Media never touches the database. It is processed into sizes, stored, and served from a CDN at the resolution each device needs.

THE HYBRID FANOUT FOR CELEBRITIES

Writing a post into every follower's feed is great until an account has millions of followers, when one post triggers millions of writes. The fix is a hybrid: push posts into precomputed feeds for ordinary accounts, but for a handful of celebrities pull their recent posts in at read time and merge them into the feed. This keeps the common case fast while surviving the extreme case.

STEP BY STEP

How an Instagram post is created and later appears in a follower's feed.

1. A user uploads a photo, which goes to object storage and is processed into several sizes.
2. A small post record with the media id, caption, and author is written to the database.

3 For a normal account, a fanout worker writes the new post id into each follower's precomputed feed.

4 When a follower opens the app, the feed service reads their precomputed list of post ids.

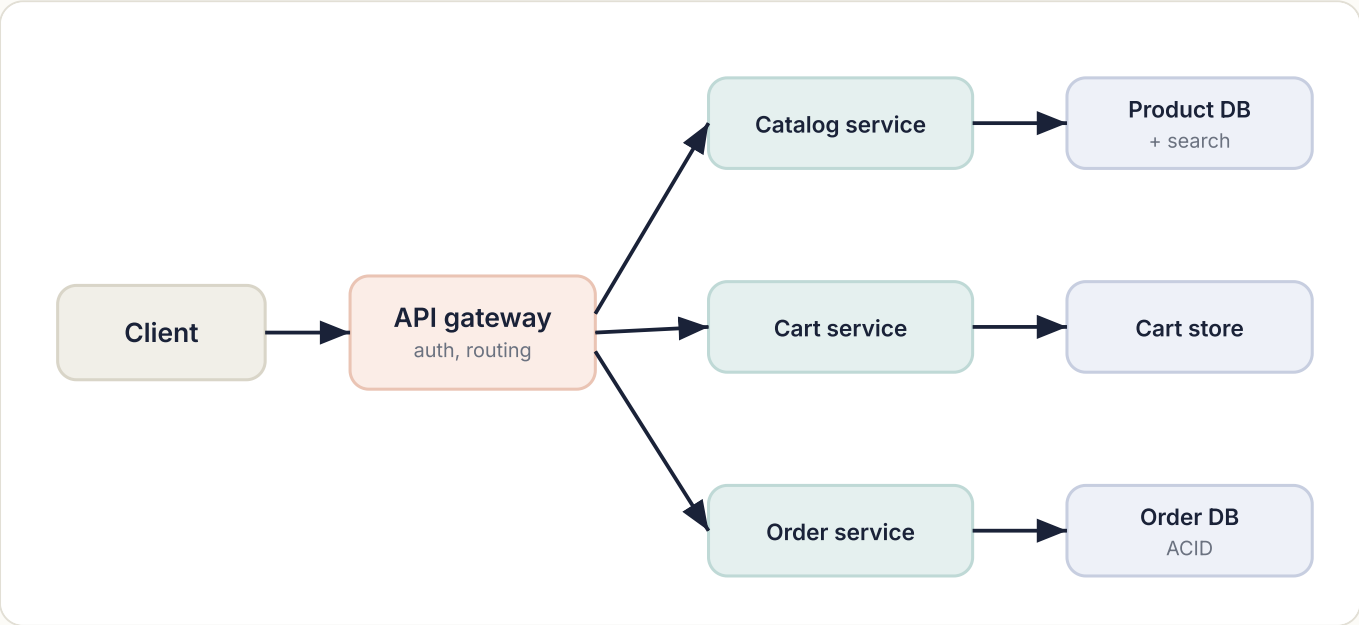
5 The app fetches each post's media from the nearest CDN edge and renders the feed.

31 Design walkthrough: Amazon

Designing an online store like Amazon is really designing several systems that cooperate: a product catalog, a shopping cart, orders, payments, and inventory. The heart of it is a tension. Browsing must be fast and always available, while checkout has to be correct down to the last unit and the last cent.

Requirements

Users browse and search a huge catalog, add items to a cart, place an order, pay, and get it shipped. Behind that, inventory must be tracked so two people cannot both buy the last item. The catalog is extremely read heavy, while orders, payments, and inventory are write sensitive and must be correct.



Each concern is its own service and store. Checkout also fans out to payment and inventory through a queue.

The key decisions

The catalog is read dominated, so cache it hard, serve it from read replicas, and back it with a search index for queries and filters. Here **eventual consistency is fine**, since a price or a rating

updating a second late hurts nobody. The cart is a fast key value store keyed by user, cheap to read and write.

Orders and payments are the opposite. They need **strong consistency and transactions**, so they sit on a relational database with ACID guarantees. The critical rule is that inventory must be reserved atomically at checkout, so the last unit is sold exactly once. Slow or non urgent steps like charging the card, sending the confirmation email, and updating recommendations are pushed onto a queue so the user's click returns immediately.

PROS

- Each service scales on its own, so the busy catalog does not slow checkout
- You can use the right store per job, cache for catalog, ACID for orders
- Queues absorb spikes and keep the checkout click fast

CONS

- Many services means more operational and networking complexity
- A cart or order spanning services needs careful coordination
- Keeping the search index in sync with the catalog adds moving parts

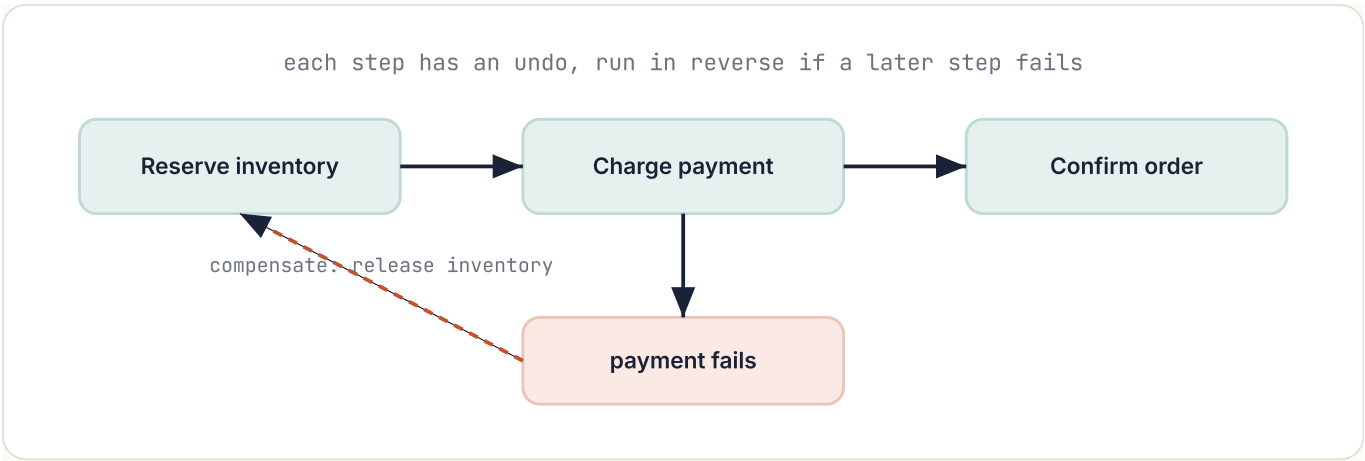
THE CONSISTENCY SPLIT IS THE WHOLE TRICK

Notice how one system uses two different consistency levels on purpose. Browsing leans available and eventually consistent because stale by a second is harmless. Checkout leans strongly consistent because overselling the last unit is not. Naming that split, and where you draw the line, is exactly the judgment an interviewer wants to see.

GOING DEEPER

Checkout as a saga across services

A checkout spans several independent services, and you cannot wrap them all in one database transaction. The saga pattern runs the steps in sequence, and gives each step a compensating action that cleanly undoes it if a later step fails. Reserve the inventory, then charge the payment, then confirm the order. If the payment fails, the compensating action releases the reserved inventory so the item is not left stuck as unavailable. This gives you correctness across services without a global lock, at the cost of writing the undo logic for every step.



A saga runs the steps in order, and if one fails it triggers compensating actions to undo the earlier ones.

THE LAST UNIT UNDER CONCURRENCY

The scary case is two customers checking out the last item at the same instant. A plain read then write can let both succeed and oversell. The reservation must be atomic, a single conditional decrement that only succeeds if stock remains, so exactly one of the two wins and the other is told it is out of stock.

STEP BY STEP

How a checkout flows through the store, keeping the last unit correct.

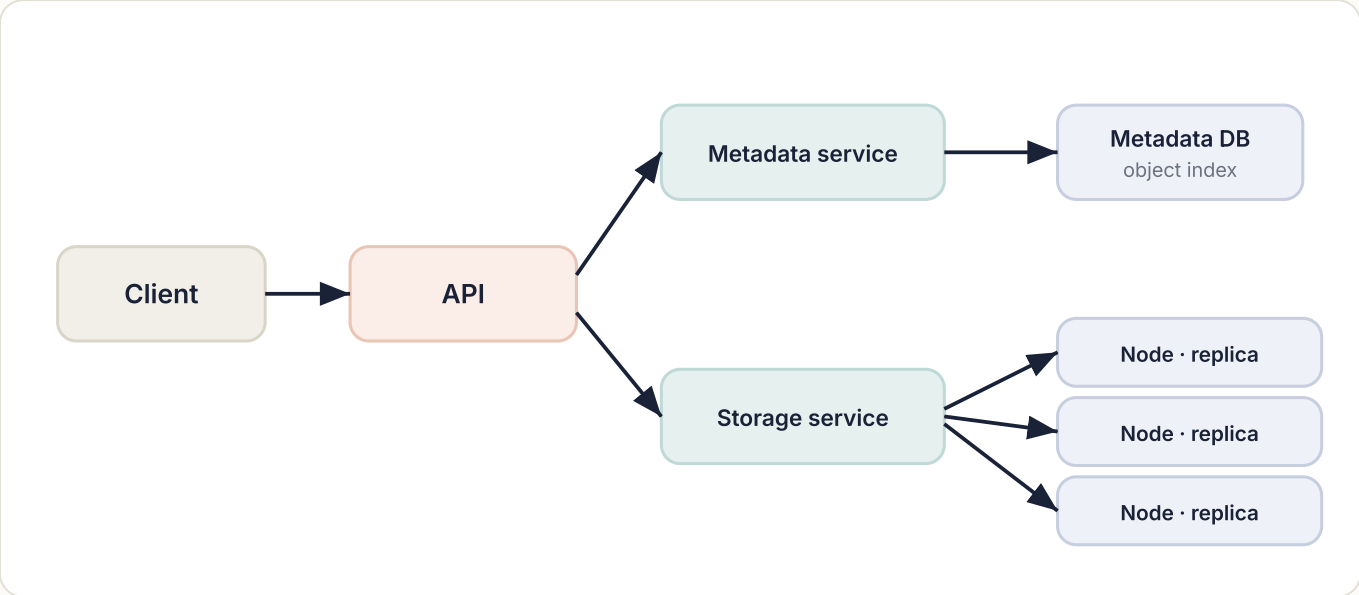
1. The user adds items to a cart held in a fast key value store.
2. At checkout, the order service begins a database transaction.
3. It reserves the inventory atomically, so the last unit in stock cannot be sold to two people.
4. It charges the payment, then records the completed order in the relational database.
5. Slower follow ups like the confirmation email and recommendation updates are pushed onto a queue, so the checkout click returns fast.

32 Design walkthrough: Amazon S3

S3 is a service that stores files, called objects, and simply never loses them. That is the whole game: durability at massive scale. Store trillions of objects across cheap, failure prone machines while guaranteeing that a dead disk, or even a whole dead building, never loses your data.

Requirements

Put and get objects by a key, group them into buckets, store objects of any size, and offer extremely high durability and availability with effectively unlimited capacity. Reads and writes are simple by key, but the durability bar is the hard part.



Small metadata is stored and queried separately from the object bytes, which are copied across many nodes.

The key decisions

The first move is to **separate metadata from the data**. Metadata (where an object lives, its size, its permissions) is small and queried constantly, so it lives in a fast, sharded store. The object bytes are huge and streamed, so they go to dedicated storage nodes. These two have completely different access patterns, and splitting them lets each scale on its own.

Durability comes from **redundancy across failure domains**. You either keep several full copies (replication) or use erasure coding, which splits an object into fragments plus parity so the whole

thing can be rebuilt from a subset. Either way the pieces are spread across separate disks, racks, and data centers, so no single failure loses data. A background process constantly watches for failed disks and re creates the lost copies. On consistency, S3 today offers strong read after write for new objects, a place where the classic CAP trade off shows up directly.

PROS

- Replication is simple and makes reads fast from any copy
- Erasure coding gives the same durability using far less storage
- Separating metadata from data lets each scale independently

CONS

- Replication multiplies storage cost by the number of copies
- Erasure coding is cheaper but costs more CPU to rebuild on read
- Re replicating after failures uses real background bandwidth

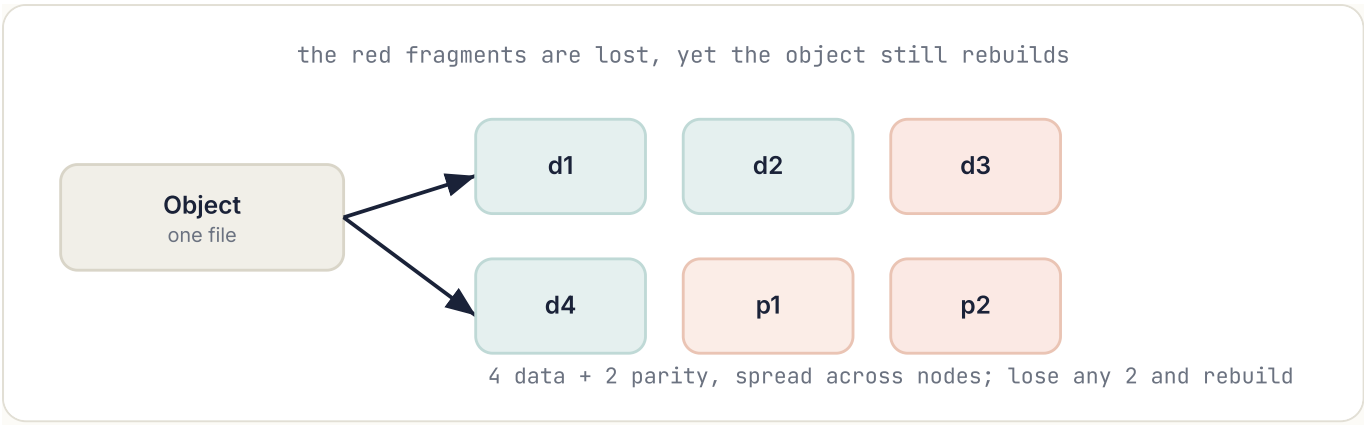
ERASURE CODING IN ONE LINE

Instead of storing three full copies, split the object into say ten data fragments and add four parity fragments, then scatter all fourteen. You can lose any four fragments and still rebuild the object perfectly. You get durability similar to replication while storing far less than three full copies, which is why huge storage systems lean on it.

GOING DEEPER

Erasure coding, and durability math

Keeping three full copies is simple but triples your storage. Erasure coding does better. You split an object into k data fragments and compute m parity fragments, then scatter all k plus m across different nodes. You can lose any m of them and still rebuild the object exactly, so you reach durability similar to replication while storing far less than three copies. Spreading the fragments across separate racks and data centers is what lets a system quote durability like eleven nines, because the chance of losing more than m fragments at the same moment becomes astronomically small.



Erasure coding splits an object into data and parity fragments, so it survives losing several of them.

PROS

- Replication is simple, and any copy serves a read instantly
- Erasure coding reaches the same durability at much lower storage cost

CONS

- Replication multiplies storage by the number of copies
- Erasure coding costs extra CPU and network to rebuild on read

---

STEP BY STEP

How storing one object keeps it safe from any single failure.

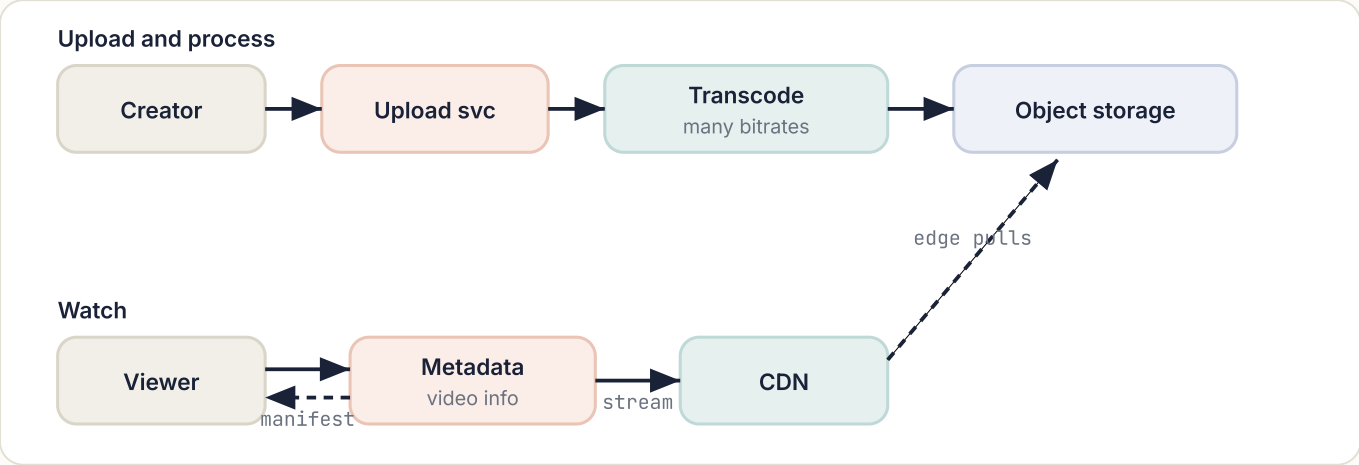
1. The client sends a put request with the object bytes and a key.
2. The service records the object's metadata, its key, size, and location, in the metadata store.
3. It splits the bytes and writes copies, or erasure coded fragments, across several nodes in different failure domains.
4. It confirms success only once enough copies are safely stored.
5. A background process keeps watching for failed disks and re creates any lost copies automatically.

33 Design walkthrough: YouTube

Streaming video is a storage and delivery problem far more than a compute one. A creator uploads one enormous file, and millions of viewers on different devices and wildly different network speeds all need it to play smoothly. Two ideas carry the design: transcoding and a CDN.

Requirements

Upload a video, process it, store it, and stream it to many devices at many quality levels, with a fast start and no buffering. Reads (views) vastly outnumber writes (uploads), and the files are huge, which pushes everything toward storage and delivery rather than databases.



One upload is transcoded into many qualities and stored, then streamed from a nearby CDN edge.

The key decisions

The core trick is **adaptive bitrate streaming**. On upload, the video is transcoded once into many resolutions and bitrates, and split into small chunks. The player then switches quality on the fly based on the viewer's current bandwidth, which is why a weak connection drops to blurry instead of freezing. Doing this heavy work upfront, at upload time, keeps playback cheap and smooth for everyone afterward.

The stored chunks live in **object storage** and are served through a **CDN**, so almost every viewer streams from a nearby edge rather than the origin across the world. This is what makes start times fast and keeps the origin from melting. The database only holds small records like the title, view count, and the list of available renditions. The heavy bytes never touch it.

PROS

- Transcoding once upfront makes every later view smooth and cheap
- A CDN gives low latency starts and offloads the origin
- Chunking lets the player adapt quality and resume instantly

CONS

- Transcoding into many formats is expensive compute per upload
- Storing every rendition multiplies storage for popular videos
- Keeping global CDN caches warm and fresh adds complexity

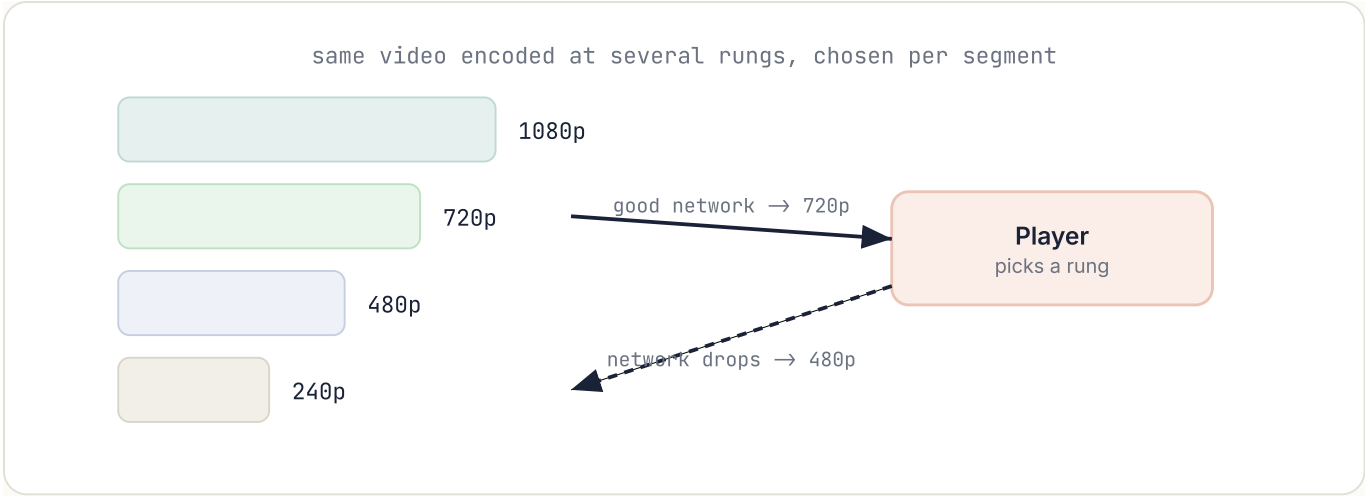
WHY THE VIDEO GETS BLURRY INSTEAD OF FREEZING

Because the same video exists at several bitrates and is cut into short chunks, the player measures your bandwidth chunk by chunk and requests a lower quality version when the network slows down. You keep watching without a pause, just at lower resolution, and it jumps back up when your connection recovers. That is adaptive bitrate streaming in action.

GOING DEEPER

Adaptive bitrate, segment by segment

On upload, the video is transcoded into a ladder of versions at different resolutions and bitrates, and each version is cut into short segments of a few seconds. The player fetches a manifest that lists the ladder, then requests one segment at a time while measuring your bandwidth as it goes. When the network slows it steps down to a lower rung for the next segment, and steps back up when it recovers, so playback keeps flowing smoothly instead of freezing. The segments are cached in a hierarchy, edge first and origin last, so popular videos are served close to viewers.



The video exists at several quality rungs. The player climbs down a rung when the network slows, so it never freezes.

THE CDN CACHE HIERARCHY

A viewer's request first hits a nearby edge cache. On a miss it goes to a larger regional cache, and only a miss there reaches the origin. Because most viewers watch the same popular videos, the edges serve the vast majority of traffic, the regional caches catch the rest, and the origin is touched rarely, which is how one video reaches millions without melting the source.

STEP BY STEP

How one upload becomes smooth playback for millions of viewers.

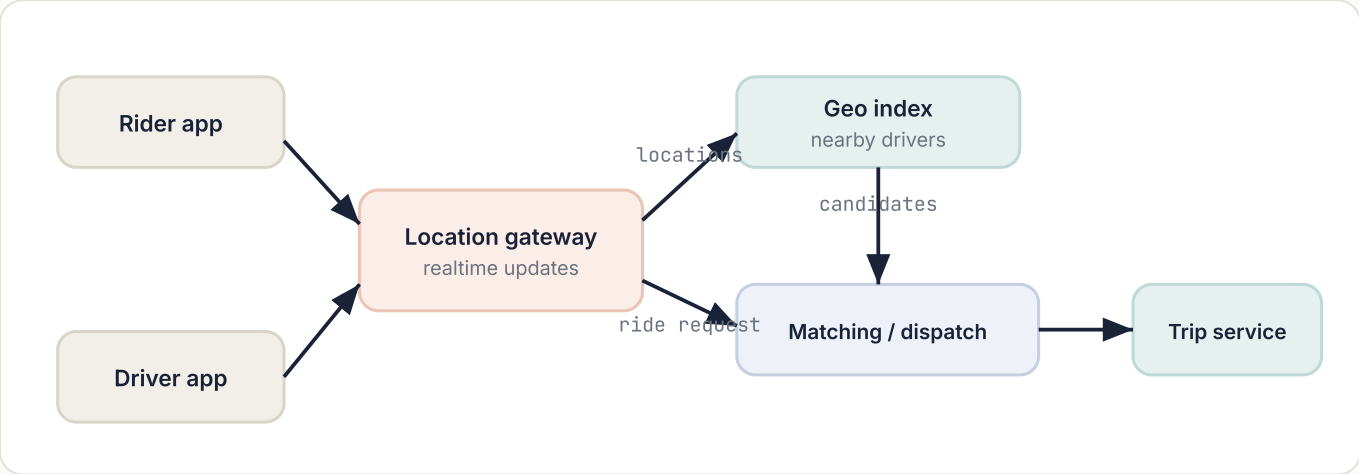
1. A creator uploads one large video file to the upload service.
2. A transcoding pipeline processes it into several resolutions and bitrates, each split into small chunks.
3. The chunks are stored in object storage, and the metadata records which renditions exist.
4. A viewer's player fetches the manifest, then streams chunks from the nearest CDN edge.
5. The player watches the network and switches quality up or down chunk by chunk, so playback never freezes.

34 Design walkthrough: Uber

A ride hailing app is a real time matching problem laid over a map. Thousands of drivers move around a city sending their location, and the moment a rider requests a ride, the system has to find a good nearby driver and connect the two within seconds.

Requirements

Track driver locations in real time, let a rider request a ride, match them to the best nearby available driver, and manage the trip from pickup to drop off and payment. The matching must be fast even with huge numbers of drivers moving at once.



Drivers stream location into a geospatial index. A request queries nearby cells and dispatches a driver.

The key decisions

The central idea is a **geospatial index**. You cannot scan every driver in the city for each request, so you divide the map into small cells using a scheme like a grid, geohash, or H3. Now "find drivers near this point" becomes a quick lookup of a handful of nearby cells instead of a full scan. Drivers push frequent location updates, and you keep only their latest position in a fast in memory store keyed by cell, since a position from ten seconds ago is worthless.

When a rider requests a ride, the **matching service** pulls the candidate drivers from the nearby cells and picks one by distance, estimated arrival time, and rating, then dispatches and holds that driver

so two riders never get the same one. Everything downstream, trip updates, pricing, and payment, flows through queues so the fast matching path is not blocked by slower work.

PROS

- Cell based indexing turns a full scan into a tiny lookup
- Keeping only the latest position in memory keeps it fast and small
- Queues keep pricing and payments off the time critical match path

CONS

- Cell size is a tuning problem, too big or too small both hurt
- Dense areas create hot cells that need extra care
- Holding a driver during dispatch needs careful concurrency

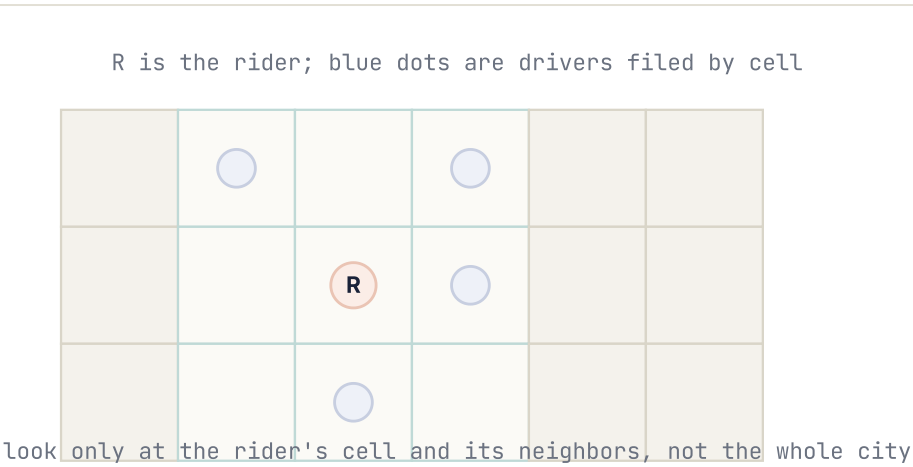
WHY A GEOSPATIAL INDEX, NOT A DATABASE QUERY

A plain database asking for every driver within two kilometers would scan and compute distances across the whole fleet on every request, which does not survive a busy city. By bucketing drivers into map cells and updating positions in memory, the system answers "who is near here" by reading a few cells. That single choice is what makes real time matching possible at scale.

GOING DEEPER

Geospatial indexing up close

The map is divided into cells using a scheme like a grid, a geohash, or Uber's H3 hexagons, and each driver's latest position is filed under its cell. To find drivers near a rider you look at the rider's cell and its immediate neighbors, a handful of cells, rather than scanning every driver in the city. Cell size is the key tuning knob: too large and each cell holds too many drivers to sort through, too small and you have to check many cells for one query. Dense downtown areas can become hot cells that need extra care, such as finer subdivision.



Drivers are filed by map cell, so finding nearby ones means checking a handful of cells instead of scanning everyone.

MATCHING UNDER CONCURRENCY

Once you have candidate drivers, two riders must not be handed the same one. The match has to atomically claim a driver, marking them busy the instant they are dispatched, so a second request sees them as taken. Without that lock you get double bookings, which is why dispatch holds a driver rather than merely suggesting one.

STEP BY STEP

How a ride request finds a nearby driver in seconds.

1. Driver apps continuously stream their location into the geospatial index, bucketed by map cell.
2. A rider requests a ride from their current location.
3. The matching service looks up the nearby cells and gathers the candidate drivers in them.
4. It picks the best one by distance, arrival time, and rating, then dispatches and holds that driver.
5. The trip service tracks the ride to completion, with pricing and payment handled through queues.

PART THREE

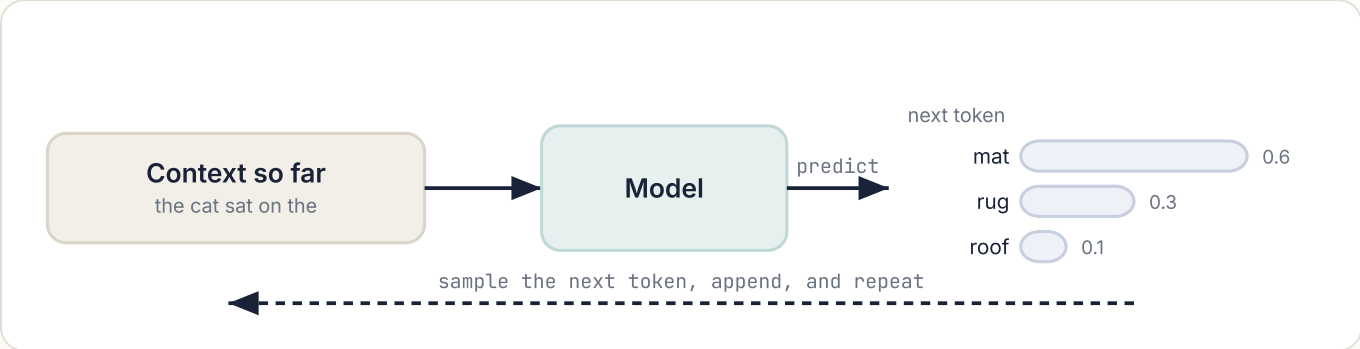
AI Engineering

How modern AI systems are actually built on top of large language models. We start with how a model works and how to prompt it, then add knowledge with RAG and memory, then turn a plain model into an agent that uses tools through MCP and reusable skills. Every idea comes with a diagram and a small worked example.

FOUNDATIONS

35 How large language models work

A large language model is, at its heart, a very good next word guesser. Everything impressive it does, answering questions, writing code, reasoning through a problem, comes from one simple loop: predict the most likely next token given everything so far, then do it again. Understanding that single idea takes the magic and the fear out of the rest.



A model reads the tokens so far, predicts the next one, you sample it, append, and loop.

The core ideas

Text is broken into **tokens**, which are word pieces, not letters. The word "unhappiness" might become three tokens. The model reads tokens and writes tokens. Given the sequence so far, it produces a probability for every possible next token, and you **sample** one, add it to the sequence, and repeat. That is the whole engine.

Two knobs matter early. The **context window** is the fixed number of tokens the model can see at once. Your system instructions, the conversation, and any documents you paste all have to fit inside it, which makes the context window the single biggest practical constraint you design around. **Temperature** controls how randomly you sample: low temperature stays focused and repeatable, high temperature gets more varied and creative.

A minimal call

You talk to a model with a list of messages, each with a role. The system message sets behavior, the user message is the request, and the model replies as the assistant.

PYTHON

```python
from anthropic import Anthropic

client = Anthropic()
resp = client.messages.create(
    model="claude-sonnet-4-5",
    max_tokens=500,
    system="You are a concise, helpful tutor.",   # sets behavior
    messages=[
        {"role": "user", "content": "Explain what a token is, in one
sentence."}
    ],
)
print(resp.content[0].text)
```

WHY MODELS HALLUCINATE

A model predicts plausible text, not verified truth. When it does not know something, it will often produce a confident, well formed answer that is simply wrong, because a fluent guess is exactly what next token prediction rewards. This is why grounding the model in real data, which the next chapters cover, matters so much.

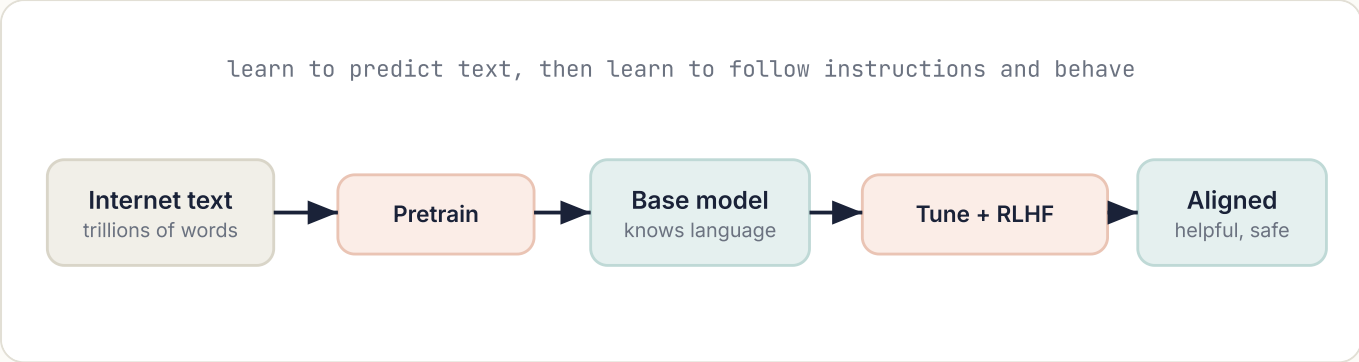
THE CONTEXT WINDOW IS YOUR BUDGET

Think of every token in the context as costing money and attention. Long conversations, big documents, and verbose instructions all compete for the same fixed space. Almost every technique later in this part, retrieval, memory, summarization, is really about spending that budget wisely.

GOING DEEPER

Where the model's ability comes from

The reason a model can write and reason is a two stage upbringing. In pretraining it reads an enormous amount of text and learns only to predict the next token, which forces it to absorb grammar, facts, and patterns of reasoning as a side effect. That produces a raw base model that is knowledgeable but not especially helpful. Fine tuning on curated examples, followed by reinforcement learning from human feedback, then teaches it to follow instructions, stay on task, and avoid harmful output. The mechanism that lets it use context at all is attention, which lets each token look back at the earlier tokens that matter most.



A model is grown in stages: pretraining teaches language, then fine-tuning and RLHF teach it to be helpful and safe.

Sampling choice	What it does
Greedy	always take the single most likely token, deterministic
Temperature	flatten or sharpen the distribution, higher is more varied
Top-p	sample only from the smallest set of tokens covering p of the probability

STEP BY STEP

What actually happens when a model generates a reply, one token at a time.

1. Your text is broken into tokens, the word pieces the model works with.
2. The model reads all the tokens and produces a probability for every possible next token.

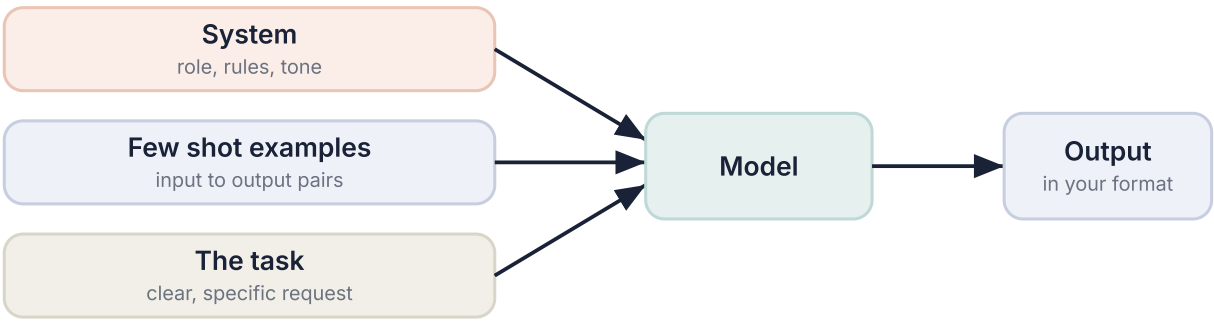
3 One token is sampled from that distribution, with temperature deciding how adventurous the choice is.

4 The chosen token is appended to the sequence.

5 Steps two through four repeat on the slightly longer sequence, until a stop token appears or the length limit is hit.

36 Prompting patterns

The prompt is your main control surface. Because the model just continues the most likely text, how you ask shapes what you get, often dramatically. A small set of patterns covers most of what actually works in practice.



A good prompt stacks a role, a few examples, and a clear task, and asks for a format you can parse.

The patterns that pay off

Be **specific and give context**, since a vague request produces a vague answer. Use **few shot examples**, a couple of input to output pairs, so the model copies the pattern you want. For anything that needs reasoning, ask for **chain of thought**, letting the model work step by step before the final answer, which measurably improves accuracy. When you will parse the result in code, ask for **structured output** like JSON that matches a schema. And set a **role and hard constraints** up front so the model stays in bounds.

Few shot plus structured output

PYTHON

```python
system = """You label a support message as one of: billing, bug, feature.
Reply with ONLY a JSON object like {"label": "..."}. No prose."""

examples = [
    {"role": "user", "content": "I was charged twice this month."},
    {"role": "assistant", "content": '{"label": "billing"}'},
    {"role": "user", "content": "The export button does nothing."},
    {"role": "assistant", "content": '{"label": "bug"}'},
]

new = [{"role": "user", "content": "Can you add a dark mode?"}]

resp = client.messages.create(
    model="claude-sonnet-4-5", max_tokens=50,
    system=system, messages=examples + new,
)
# → {"label": "feature"}
```

ASK FOR THE FORMAT YOU WILL PARSE

If your code needs JSON, say so explicitly, show an example of the exact shape, and tell the model to return only that with no extra prose. Models are good at following a format when you make it concrete, and it saves you from brittle text parsing later.

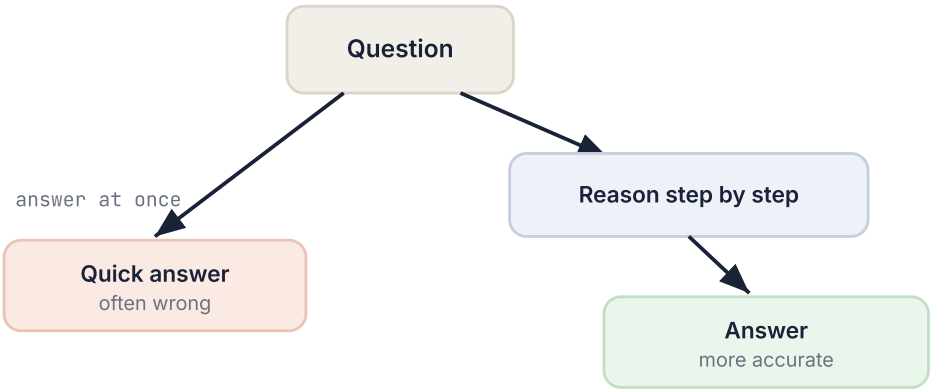
MORE IS NOT BETTER, RELEVANT IS BETTER

Stuffing a prompt with everything you can think of often hurts. Extra, loosely related text dilutes the important parts and burns your context budget. Give the model exactly what it needs for this task, clearly organized, and nothing more.

GOING DEEPER

Why chain of thought actually helps

A model does a fixed amount of computation per token, so when you demand an answer immediately it has to leap straight to the result with no room to work. Asking it to reason step by step lets it spend tokens laying out intermediate results, and each of those becomes context the next step can build on, which is why accuracy on multi step problems jumps. The same logic explains few shot prompting: the examples you show become part of the context the model continues from, steering its output toward the pattern you demonstrated.



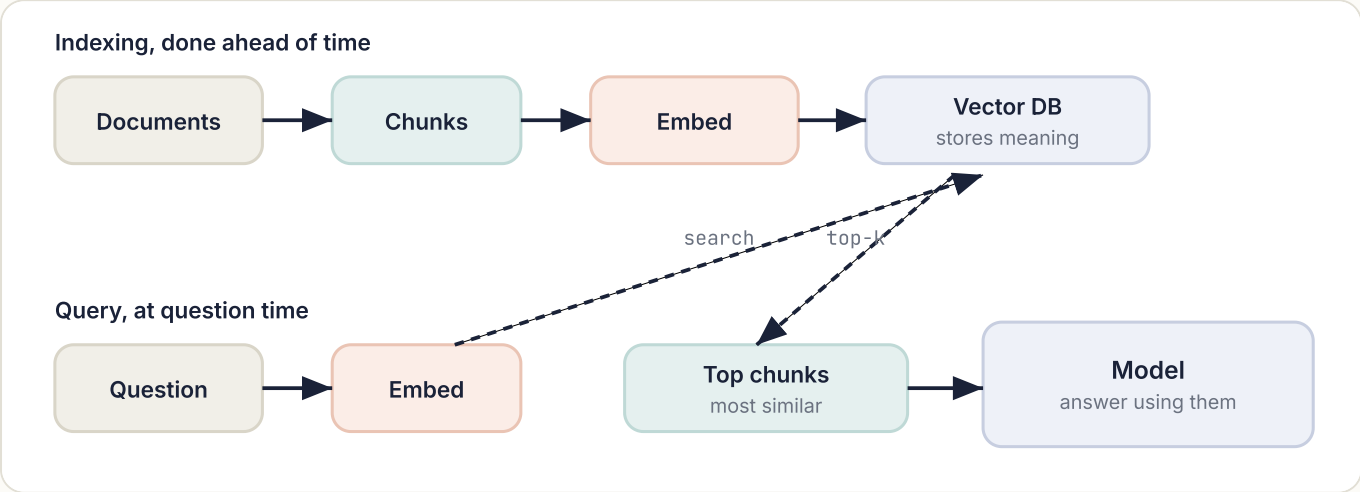
Letting the model work through intermediate steps in tokens gives it room to compute, which lifts accuracy on hard questions.

Technique	When to reach for it
Zero-shot	simple, well known tasks where a clear instruction is enough
Few-shot	when the format or style is easier to show than describe
Chain of thought	multi step reasoning, math, and logic
Structured output	whenever your code will parse the result

KNOWLEDGE AND MEMORY

37 RAG, retrieval augmented generation

A model only knows what was in its training data and what fits in the context window. RAG fixes both. It fetches the most relevant pieces of your own data at question time and drops them into the prompt, so the model answers from fresh, private knowledge instead of guessing, and can even cite its sources.



Documents are chunked and embedded ahead of time. A question retrieves the closest chunks and feeds them to the model.

How it works, in two phases

First, **indexing**, done ahead of time. You split your documents into **chunks**, turn each chunk into an **embedding** (a vector of numbers that captures its meaning), and store those vectors in a **vector database**. Chunks with similar meaning end up close together in that vector space.

Then, at **query** time, you embed the user's question the same way, search the vector database for the chunks whose vectors are closest to it, and paste those chunks into the prompt as context. The model then answers using that retrieved text. The quality of what you retrieve decides the quality of the answer, so retrieval is the heart of RAG.

PYTHON

```python
def answer_with_rag(question, vector_db, client):
    q_vec = embed(question)                          # same embedder as
indexing
    chunks = vector_db.search(q_vec, top_k=5)      # nearest chunks by
meaning
    context = "\n\n".join(c.text for c in chunks)

    prompt = f"""Answer using ONLY the context below. Cite the source ids.
Context:
{context}

Question: {question}"""

    resp = client.messages.create(
        model="claude-sonnet-4-5", max_tokens=500,
        messages=[{"role": "user", "content": prompt}],
    )
    return resp.content[0].text
```

EMBEDDINGS AND VECTOR SEARCH, PLAINLY

An embedding turns a piece of text into a point in a high dimensional space where nearness means similar meaning. So "how do I reset my password" lands near a support article about password resets, even if they share no exact words. Vector search is just finding the nearest points to your question, which is why RAG retrieves by meaning rather than keyword.

PROS

- Answers from fresh, private data the model never trained on
- Fewer hallucinations, since the model is grounded in real text
- You can show citations back to the source chunks
- Cheaper and faster to update than retraining the model

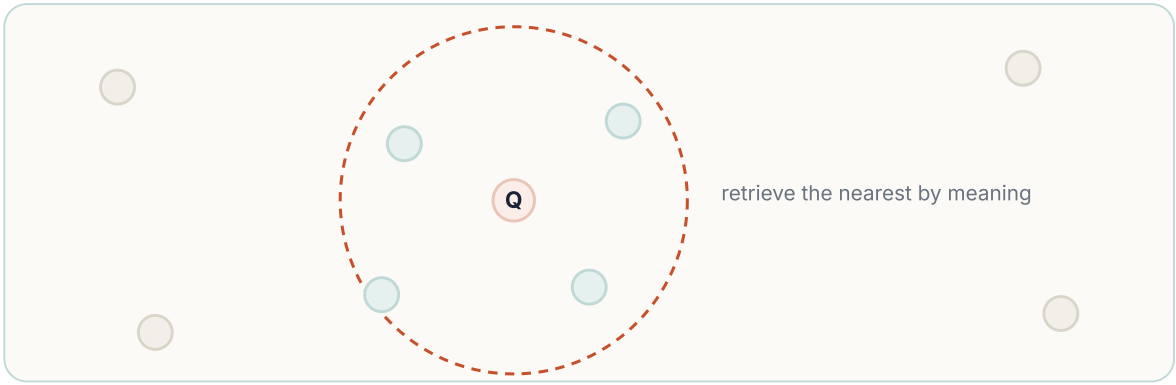
CONS

- Bad retrieval means a confident but wrong answer
- Chunking and embedding choices strongly affect quality
- Retrieved text still competes for the context budget
- Adds a vector database and a pipeline to maintain

GOING DEEPER

Why retrieval quality is the whole game

RAG lives or dies on what you retrieve, because the model can only answer from the chunks you hand it. Two levers dominate. Chunking decides how documents are split: chunks too large dilute the relevant sentence, too small lose context, so a sensible size with a little overlap matters a lot. Retrieval decides what comes back: pure vector search finds semantic matches, but combining it with keyword search (hybrid search) and then re ranking the top results with a stronger model catches cases where meaning and exact terms both matter. Get retrieval right and the generation step is easy; get it wrong and no prompt can save the answer.



Q = question, teal = relevant chunks nearby, grey = irrelevant chunks far away

Embeddings place text so that similar meaning sits close together, so retrieval is just finding the nearest points to the question.

	RAG	Fine-tuning
Best for	fresh or private facts, citations	changing style, format, or behavior
Update cost	cheap, just re index	expensive, retrain
Risk	bad retrieval gives wrong facts	can forget or drift

STEP BY STEP

RAG has two phases: build the index once, then use it on every question.

Indexing, once

1. Split your documents into chunks.
2. Turn each chunk into an embedding vector.
3. Store the vectors in a vector database.

Answering, per question

1. Embed the user's question with the same embedder.
2. Search the vector database for the closest chunks by meaning.
3. Paste those chunks into the prompt as context.
4. Ask the model to answer using only that context, and return the answer with citations.

38 Memory

By default a model remembers nothing between calls. It is completely stateless. Any sense of memory is something you build by choosing what to carry back into the context window. Good memory design is really just good context management.

Memory is built, not given. Summarize and save what matters, then retrieve only the relevant pieces later.

The kinds of memory

Working memory is just the current conversation sitting in the context window. It is immediate but limited, and it disappears when the window fills or the session ends. **Long term memory** is anything you store outside the model and pull back in when it is relevant, which uses the very same embedding and retrieval idea as RAG, only now the documents are your own past interactions and facts.

People often split long term memory into **episodic** (things that happened, like past conversations), **semantic** (stable facts and preferences, like the user's name or timezone), and **procedural** (how to do things, which overlaps with skills later in this part). You do not need all of them, but naming them helps you decide what is worth keeping.

The practical loop

As a conversation grows, you **summarize** older turns so they take fewer tokens, and you **extract** durable facts worth remembering. You save those to a store. On a later turn, you **retrieve only the memories relevant** to the new message and place them back into the context. The trick is selectivity: pulling everything back in wastes the budget and can even confuse the model.

PYTHON

```python
# Long term memory is just retrieval over your own history.
def remember(fact, store):
    store.add(text=fact, vector=embed(fact))

def recall(message, store, k=3):
    return store.search(embed(message), top_k=k)   # only what fits the
moment

# On each turn:
memories = recall(user_message, store)
context = format_memories(memories) + "\n" + user_message
```

MEMORY IS RETRIEVAL INTO THE CONTEXT WINDOW

There is no separate memory organ inside the model. Every technique here ends the same way: put the right text back into the context before you call the model. Once you see memory as selective retrieval into a fixed budget, designing it becomes straightforward.

GOING DEEPER

Managing the context budget

Because the context window is a fixed size, everything you might want the model to know competes for the same space: the system prompt, the running conversation, any retrieved documents, recalled memories, and the room left for the answer. Memory design is the craft of fitting the right things into that budget. The tools are summarizing older turns so they take fewer tokens, keeping only a sliding window of recent messages verbatim, and retrieving just the handful of past facts relevant to the current message instead of everything. Seen this way, there is no separate memory store inside the model, only clever choices about what to place in the window before each call.

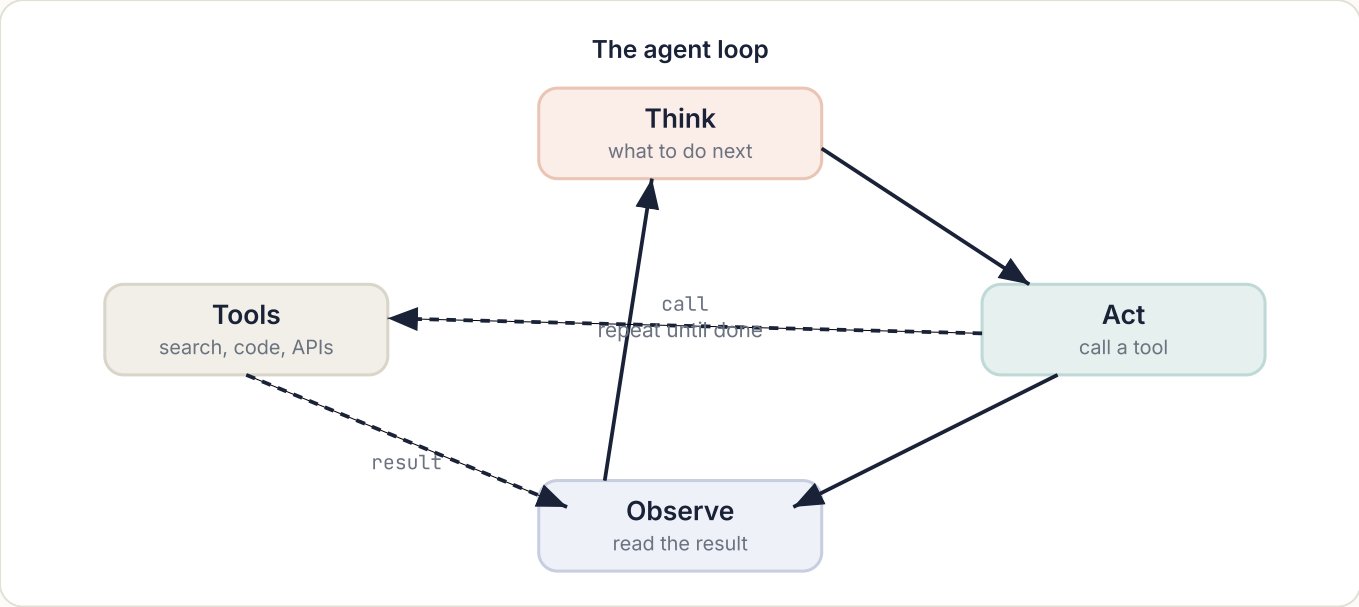
Everything competes for one fixed window, so memory design is really about spending that budget wisely.

Memory type	What it holds
Episodic	things that happened, like past conversations
Semantic	stable facts, like a user's name or preferences
Procedural	how to do things, which overlaps with skills

AGENTS AND TOOLS

39 Agentic patterns

A plain model just answers in one shot. An agent can act. It decides to use a tool, looks at the result, and keeps going until the job is done. That loop, plus a few patterns layered on top, is what turns a chatbot into something that books the flight, fixes the code, or researches the report.



An agent loops: reason about the next step, act with a tool, observe the result, and repeat until finished.

The core loop

The workhorse pattern is often called **think, act, observe**. The model **reasons** about what to do next, **acts** by emitting a structured tool call, and your code runs that tool and feeds the result back so the model can **observe** it. Then it loops. It might search the web, run code, or hit an API, using each result to decide the next step, until it has enough to give a final answer.

The patterns on top

Tool use, also called function calling, is the foundation: the model outputs which tool to call and with what arguments, and you execute it. **Planning** has the agent break a goal into steps first, then carry them out. **Reflection** has it critique and revise its own output before finishing. **Multi agent** setups split the work across specialists, say a planner, a coder, and a reviewer, that hand work to each other.

PYTHON

```
def run_agent(goal, tools, client):
    messages = [{"role": "user", "content": goal}]
    for step in range(MAX_STEPS):                 # always cap the loop
        resp = client.messages.create(
            model="claude-sonnet-4-5", max_tokens=1000,
            tools=tools, messages=messages,
        )
        messages.append({"role": "assistant", "content": resp.content})

        if resp.stop_reason ≠ "tool_use":
            return resp                            # model gave a final
answer

        for call in tool_calls(resp):              # act: run each requested
tool
            result = run_tool(call.name, call.input)
            messages.append(tool_result(call.id, result))   # observe
    raise RuntimeError("hit the step limit")
```

THE MODEL DECIDES, YOUR CODE EXECUTES

A crucial split: the model only proposes a tool call, it never runs anything itself. Your code decides whether to run it, validates the inputs, executes, and returns the result. That boundary is where you put guardrails, permissions, and logging, and it is what keeps an agent safe.

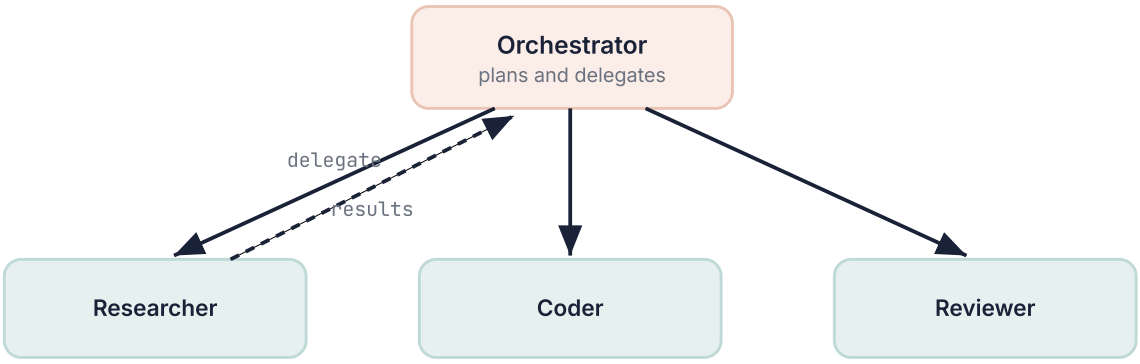
LOOPS CAN RUN AWAY

An agent that keeps thinking and acting can loop forever, rack up cost, or wander down a wrong path. Always cap the number of steps, validate what tools return, and for anything risky like sending email or spending money, keep a human or a hard rule in the loop.

GOING DEEPER

Beyond the basic loop

The think, act, observe loop is the foundation, but a few patterns layer on top for harder work. Plan and execute has the agent write a full plan first, then carry out each step, which keeps it from wandering. Reflection has it critique and revise its own output before finishing, catching its own mistakes. Multi agent setups split a big job across specialists, say a researcher, a coder, and a reviewer, coordinated by an orchestrator, so each agent has a narrow, well defined role. More structure is not always better, though, so reach for these only when a single loop genuinely struggles.



specialists hand work back and forth under one orchestrator

For bigger jobs, an orchestrator splits the work across specialist agents that each focus on one thing.

Pattern	Use it when
ReAct loop	most tasks, tool use with reasoning
Plan and execute	long tasks that benefit from a plan up front
Reflection	quality matters and self review helps
Multi-agent	distinct subtasks need distinct specialists

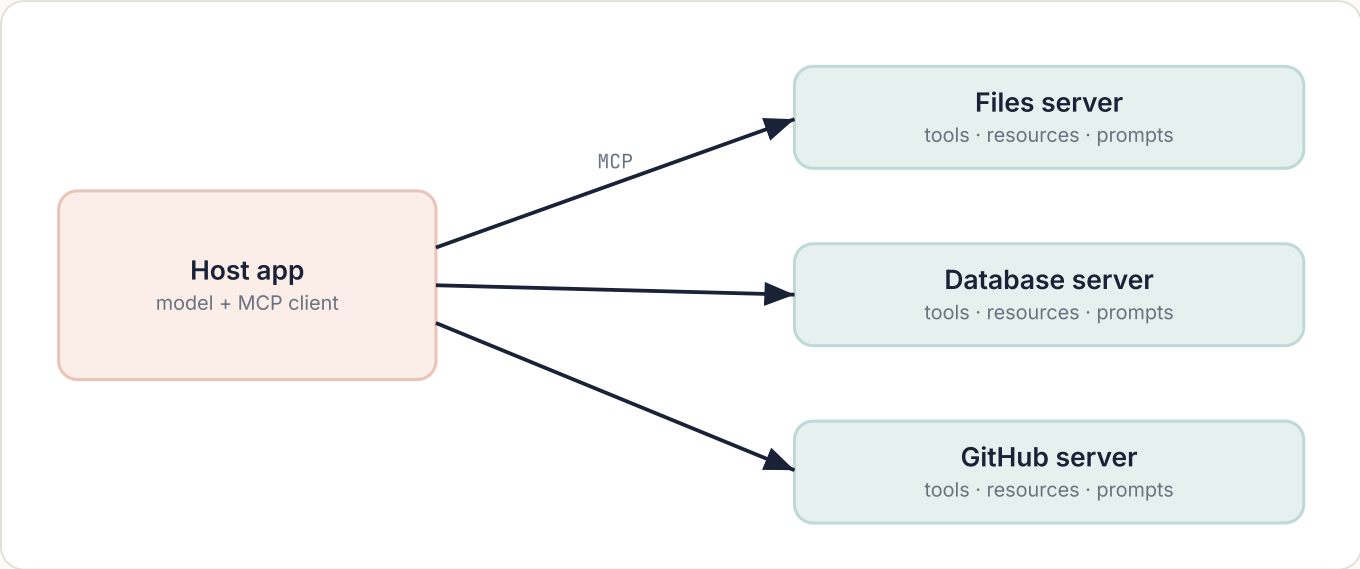
STEP BY STEP

One turn of the agent loop, from reasoning to the next action.

1. The model reads the goal and everything observed so far.
2. It reasons about the next step and decides whether it needs a tool.
3. If it does, it emits a structured tool call with the arguments it wants.
4. Your code validates and runs that tool, then feeds the result back in.
5. The model observes the result and loops again, or returns a final answer if the task is done. A step limit stops any runaway loop.

40 MCP, the Model Context Protocol

Once agents start using tools, every application ends up reinventing how to connect a model to data and actions. The Model Context Protocol is an open standard that fixes this. It is a common way for an AI app to plug into external tools, data, and prompts, a bit like a universal port for AI, so any compatible app can use any compatible integration.



One standard way for an AI app to plug into many external tools, data sources, and prompt templates.

How it fits together

The **host** is the AI application, the thing that runs the model. It contains an **MCP client**. A **server** is a small separate program that exposes some capability, and you run many of them, typically one per integration: one for your files, one for a database, one for GitHub, and so on. The client connects to these servers in a standard way and lets the model use what they offer.

Each server can expose three kinds of things. **Tools** are actions the model can call, like "create an issue" or "run a query". **Resources** are data the model can read, like a file or a record. **Prompts** are reusable prompt templates the server provides. Because the protocol is standard, the client discovers all of this automatically.

MCP SERVER

```
# A tiny MCP server exposing one tool.
from mcp.server import Server

server = Server("weather")

@server.tool()
def get_weather(city: str) → str:
    """Return the current weather for a city."""
    data = weather_api.lookup(city)
    return f"{data.temp}C and {data.summary}"

# Any MCP compatible host can now discover and call get_weather,
# no custom glue code needed in the app itself.
```

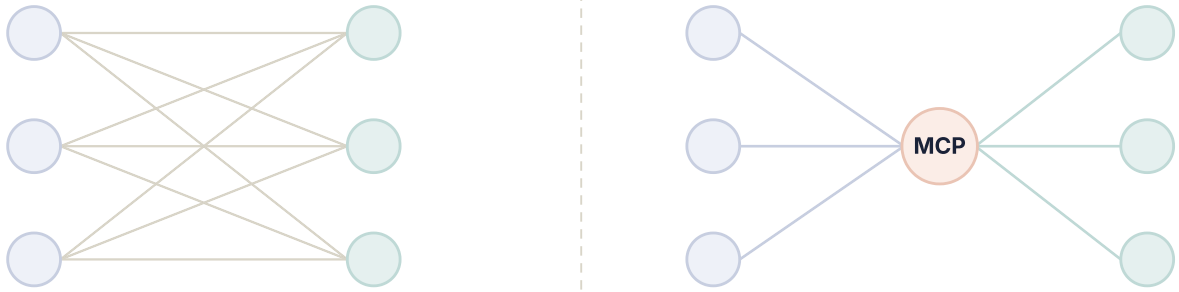
WRITE THE INTEGRATION ONCE

The big win is decoupling. You build a tool as an MCP server one time, and then every MCP compatible client can use it without custom code. Tools stop being welded into a single app and become reusable building blocks, which is exactly why the standard caught on so fast.

GOING DEEPER

The M times N problem it solves

Before a standard, connecting M applications to N tools meant building M times N custom integrations, a tangle that grows impossibly fast. MCP collapses this to M plus N: each application implements one MCP client, each tool is wrapped once as an MCP server, and any client can talk to any server. When a client connects, it asks the server what it offers and the server describes its tools, resources, and prompts, so the discovery is automatic and nothing is hard coded. That single decoupling is why an integration written once becomes usable everywhere.



without a standard: 9 custom integrations

with MCP: 3 + 3 through one standard

Without a standard, every app must integrate every tool. MCP turns that tangle into one connection per side.

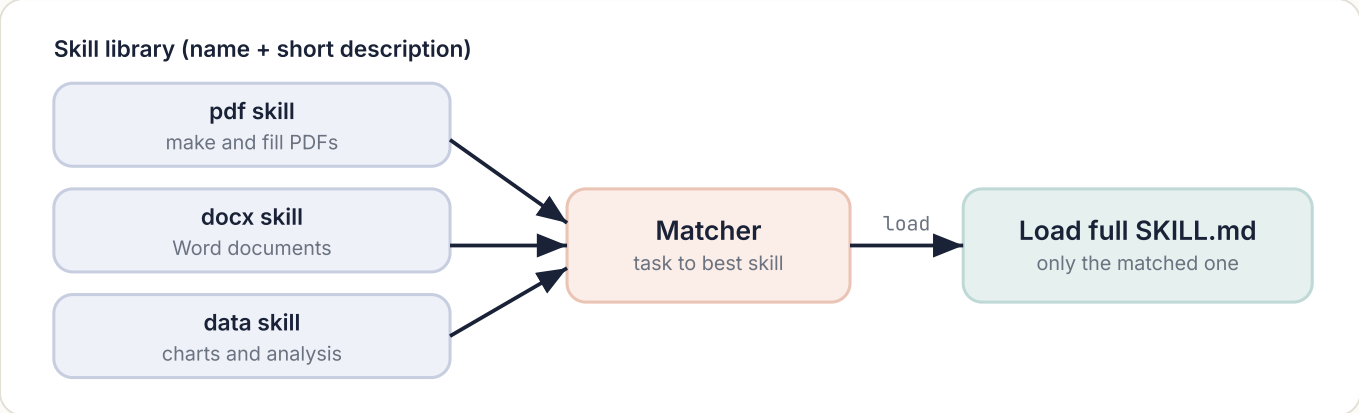
HOW A CLIENT AND SERVER ACTUALLY TALK

An MCP server can run locally over standard input and output, or remotely over HTTP with server sent events for streaming. On connecting, the client and server negotiate capabilities, then the client can list and call tools, read resources, and fetch prompt templates through the same standard messages, regardless of what the server does underneath.

41

Skills

As agents take on bigger jobs, you do not want to cram every possible instruction into one giant prompt. A skill is the answer: a self contained folder of instructions and resources that the agent pulls in only when the task actually calls for it. Reusable know how, loaded on demand.



The agent sees only short descriptions until a task matches, then loads that one skill's full instructions.

What a skill is

A skill is a folder with a **SKILL.md** file at its root. That file has a name and, crucially, a **description of when to use it**, followed by the detailed instructions and any helper scripts or templates. The description is what lets the agent decide whether this skill is relevant to the task in front of it.

Progressive disclosure

The clever part is how skills are loaded. The agent first sees only the short name and description of every skill, which costs almost nothing. When a task matches one, it loads that skill's full instructions, and only then. This **progressive disclosure** keeps the context lean while still giving the agent access to a large library of detailed procedures. Skills capture hard won, reusable methods, like how to produce a polished document or fill a form, so the agent does not have to rediscover them every time.

SKILL.MD

```
---
name: pdf
description: >
  Create, read, edit, or fill PDF files. Use whenever the user
  wants to make a PDF, extract text or tables, merge or split
  pages, add a watermark, or fill a form. Do NOT use for Word docs.
---

# Making a PDF

1. Draft the content in HTML with the house styles below.
2. Render it to PDF with the provided script.
3. Save the result to the outputs folder and share the link.

(the rest of the detailed, reusable procedure follows)
```

THE DESCRIPTION DECIDES EVERYTHING

A skill only helps if it triggers at the right moment, and that depends entirely on how well its description names the situations it applies to. A vague description gets skipped or fires at the wrong time. This very system uses skills exactly like these to build Word documents, spreadsheets, and PDFs, which is a real example of the pattern in action.

GOING DEEPER

Progressive disclosure, layer by layer

Skills work because they are revealed in layers rather than dumped into the context all at once. At the cheapest layer, the agent sees only each skill's name and one line description, which costs almost nothing even across a large library. When a task matches a description, the agent loads that skill's full instructions, the SKILL.md body. And only if the procedure calls for it does it reach the deepest layer, the bundled scripts, templates, and reference files. This layering is what lets an agent carry hundreds of skills without drowning its context, and it is exactly why a skill's description has to name its situations precisely, since that one line is what triggers everything below it.

Skills reveal themselves in layers, so the agent pays only for the detail a task actually requires.

	What it is	When it runs
Skill	reusable instructions and resources	loaded when a task matches
Tool	an action the model can call	invoked mid task for a result
RAG	facts fetched from your data	retrieved to ground an answer

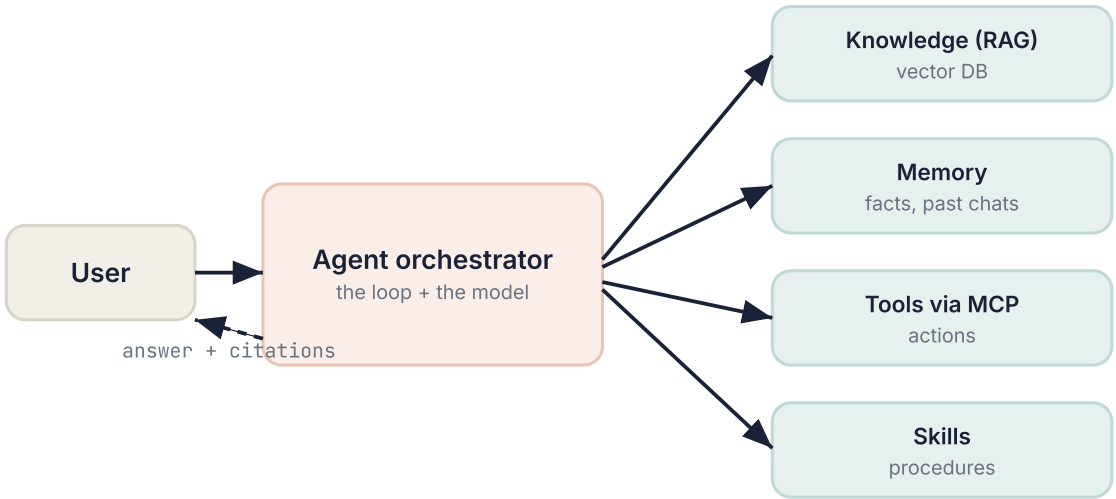
PUTTING IT TOGETHER

42 Design walkthrough: an AI assistant

Let us wire the whole part together into one system: an assistant that answers questions over your company's documents, remembers context across a conversation, and can take real actions. This is where the model, RAG, memory, the agent loop, MCP, and skills all meet.

Requirements

Answer questions from private documents with citations, hold a conversation that remembers earlier context, take actions like creating a ticket or sending an email through tools, and stay safe while doing it. Reads dominate, and correctness plus traceability matter more than raw speed.



One orchestrator runs the loop and pulls knowledge, memory, tools, and skills into the model's context as needed.

How the pieces fit

An **orchestrator** runs the agent loop and owns the context window. When a request comes in, it pulls in the relevant **memory** and any matching **skill**, then enters the loop: retrieve supporting text with **RAG**, let the model think, possibly call a **tool through MCP**, observe the result, and repeat until it can answer. It returns the answer with citations back to the source chunks, and updates memory with anything worth keeping for next time.

The decisions that matter

Keep the context lean: retrieve what you need rather than stuffing everything in, because every token competes for the same budget. Put **guardrails** on any tool that changes the world, with validation and, where needed, human approval. Add **evals**, a set of test questions with known good answers, so you can measure whether a change actually helped. And cache embeddings and frequent retrievals to keep it fast and cheap.

PROS

- Answers are grounded in real data and come with citations
- The agent can actually act, not just talk, through tools
- Memory and skills keep it consistent across a long session
- Each piece can be improved or swapped on its own

CONS

- An agent is more complex and harder to debug than a single call
- Every retrieval, tool, and skill competes for the context budget
- Actions need real guardrails, which take effort to get right
- Without evals you cannot tell if a change helped or hurt

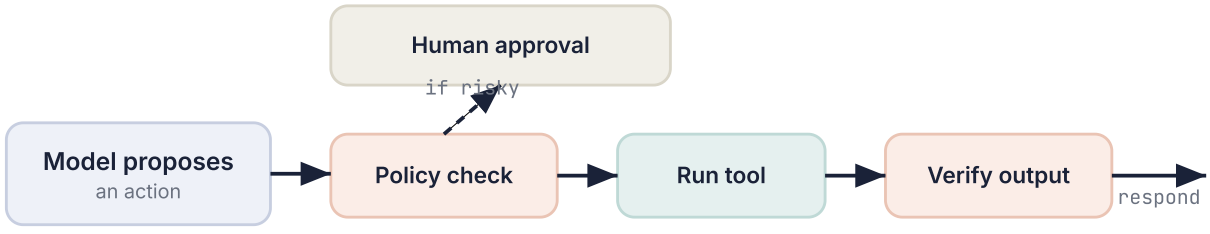
EVERYTHING COMPETES FOR THE CONTEXT WINDOW

Notice the theme running through this whole part. The system prompt, the conversation, retrieved documents, recalled memories, tool results, and loaded skills all share one fixed context budget. Designing an AI system is mostly the art of deciding, at each moment, what deserves a place in that window.

GOING DEEPER

Making it reliable: guardrails and evals

An agent that can act needs a safety layer, because the model only proposes an action, it never runs one directly. That boundary is where guardrails live: validate the proposed inputs against a policy, escalate anything risky like spending money or sending email to a human for approval, run the tool, then verify the result before continuing. Just as important is knowing whether the system is any good, which is what evals are for. You build a set of test cases with known good answers and run them on every change, so you can measure whether a tweak helped or quietly made things worse, rather than guessing.



Guardrails sit between the model's proposal and the real action, validating inputs and outputs and escalating risky steps.

HOW TO EVALUATE AN AI SYSTEM

Keep an offline test set of representative inputs with golden answers, and score each change against it. For open ended output where there is no single right answer, a common approach is to use a strong model as a judge, grading responses against a rubric. Track the numbers over time so quality becomes something you measure, not something you hope for.

STEP BY STEP

The full lifecycle of one request through the AI assistant.

1. A request arrives at the orchestrator, which owns the context window.
2. It loads the relevant long term memory and any skill whose description matches the task.
3. It retrieves supporting text from the knowledge base with RAG.

4 The model reasons, and may call a tool through MCP, whose result is observed.

5 The loop repeats until the model can answer, which it returns with citations to the sources.

6 Anything worth remembering is written back to memory for next time.