

DSA interview questions

28 questions · 9 easy · 18 medium · 1 hard. Open one to see the answer, a code example and the likely follow-up, then mark it as known or to review.

1

What is Big-O? Give the complexity of common operations.

Easy



#complexity

Big-O describes how time or memory **grows with input size** in the worst case, ignoring constants: $O(2n + 5)$ is $O(n)$.

Common operations:

- array index access $O(1)$, search $O(n)$, push/pop at end $O(1)$ amortised, insert/remove at front $O(n)$
- hash map get/set/delete $O(1)$ average, $O(n)$ worst
- binary search on a sorted array $O(\log n)$
- balanced BST or heap insert/remove $O(\log n)$, heap peek $O(1)$
- good comparison sorts $O(n \log n)$
- nested loops over the input $O(n^2)$, all subsets $O(2^n)$, all permutations $O(n!)$

Rule of thumb: about 10^8 simple operations per second, so $n = 10^5$ needs $O(n \log n)$ or better.

Likely follow-up: What does amortised $O(1)$ mean for a dynamic array push?

2

Array vs linked list: when would you use each?

Easy



#arrays #linked list

Array: contiguous memory. $O(1)$ random access by index, very cache-friendly, but inserting or deleting in the middle shifts elements, $O(n)$.

Linked list: nodes with **next** pointers scattered in memory. Insert or delete is $O(1)$ **once you have the node**, but access by index is $O(n)$ and pointer chasing is cache-unfriendly, plus extra memory per node.

In practice arrays (dynamic arrays) win almost always. Linked lists shine for queues/deques with frequent removals at both ends, LRU caches (doubly linked list + hash map), and when you splice lists together.

3

Two Sum: find indices of two numbers that add up to a target.

Easy



#hash map #arrays #coding

Brute force checks every pair: $O(n^2)$.

Better: walk the array once with a **hash map** from value to index. For each number, check whether its complement $\text{target} - x$ was already seen. If yes, return both indices; otherwise store the current number.

- Time $O(n)$, space $O(n)$
- If the array is sorted, two pointers give $O(1)$ space

This "store what you have seen in a map" trick solves a whole family of problems (subarray sum equals k, pairs with difference k).

Example

```
function twoSum(nums, target) {
  const seen = new Map()           // value -> index
  for (let i = 0; i < nums.length; i++) {
    const need = target - nums[i]
    if (seen.has(need)) return [seen.get(need), i]
    seen.set(nums[i], i)
  }
  return []
}

console.log(twoSum([2, 7, 11, 15], 9)) // [0, 1]
console.log(twoSum([3, 2, 4], 6))      // [1, 2]
console.log(twoSum([3, 3], 6))         // [0, 1]
```

? Likely follow-up: Now find all unique triplets that sum to zero (3Sum).

4

Reverse a singly linked list.

Easy



#linked list #coding

Walk the list once and flip each `next` pointer to point backwards. Keep three pointers: `prev` (already reversed part), `curr`, and `next` (saved before you overwrite `curr.next`).

At the end `prev` is the new head.

- Time `O(n)`, space `O(1)` iteratively
- The recursive version is `O(n)` space for the call stack

The key is to **save `next` before rewiring**, or you lose the rest of the list.

Example

```
const node = (val, next = null) => ({ val, next })
const fromArray = (a) => a.reduceRight((next, v) => node(v, next), null)
const toArray = (h) => { const out = []; for (; h; h = h.next)
  out.push(h.val); return out }

function reverse(head) {
  let prev = null
  let curr = head
  while (curr) {
    const next = curr.next // save
    curr.next = prev       // flip
    prev = curr
    curr = next
  }
  return prev
}

console.log(toArray(reverse(fromArray([1, 2, 3, 4, 5]))))
console.log(toArray(reverse(fromArray([]))))
```

5

Detect a cycle in a linked list (Floyd's algorithm).

Medium



#linked list #two pointers #coding

Floyd's tortoise and hare: move `slow` one step and `fast` two steps at a time. If there is no cycle, `fast` reaches `null`. If there is one, fast enters the loop and **gains one step per iteration**, so it must eventually land on slow.

- Time $O(n)$, space $O(1)$ (a hash set of visited nodes also works but uses $O(n)$ space)

To find where the cycle **starts**: after they meet, reset one pointer to the head and move both one step at a time; they meet at the cycle entrance.

The same fast/slow idea finds the middle of a list.

Example

```
function cycleStart(head) {
  let slow = head, fast = head
  while (fast && fast.next) {
    slow = slow.next
    fast = fast.next.next
    if (slow === fast) {
      slow = head
      while (slow !== fast) { slow = slow.next; fast = fast.next }
      return slow // entrance of the cycle
    }
  }
  return null
}

const nodes = [1, 2, 3, 4, 5].map((val) => ({ val, next: null }))
nodes.forEach((n, i) => (n.next = nodes[i + 1] ?? null))
console.log(cycleStart(nodes[0])) // null: no cycle
nodes[4].next = nodes[2] // 5 -> 3
console.log(cycleStart(nodes[0]).val) // 3
```

6

Valid parentheses: check that brackets are balanced.

Easy



#stack #strings #coding

Use a **stack**. Push every opening bracket. On a closing bracket, the top of the stack must be **the matching opener**; pop it, otherwise the string is invalid. At the end the stack must be empty.

- Time $O(n)$, space $O(n)$

Edge cases interviewers check: an empty string (valid), a closer with an empty stack (`)`, and leftover openers (`(( (`).

Stacks are the go-to for anything "most recent unmatched": undo, expression evaluation, next greater element (monotonic stack).

Example

```
function isValid(s) {
  const pairs = { ')': '(', ']': '[', '}': '{' }
  const stack = []
  for (const ch of s) {
    if (ch in pairs) {
      if (stack.pop() !== pairs[ch]) return false
    } else stack.push(ch)
  }
  return stack.length === 0
}
for (const s of ['()[]{}', '([{}])', ']', '([])', '(', ''])
  console.log(JSON.stringify(s), isValid(s))
```

7

Implement binary search. What is "binary search on the answer"?

Medium



#binary search #coding

On a **sorted** array, compare with the middle and discard half each step: $O(\log n)$ time, $O(1)$ space. Use $lo + \text{Math.floor}((hi - lo) / 2)$ and be precise about the loop condition and bounds to avoid infinite loops.

Binary search on the answer: when the answer is a number in a range and a check $ok(x)$ is **monotonic** (false, false, true, true...), binary search for the first x that works. Examples: minimum ship capacity to ship packages in D days, Koko eating bananas, square root.

Time: $O(\log(\text{range}) * \text{cost of } ok)$.

Example

```
function search(arr, target) {
  let lo = 0, hi = arr.length - 1
  while (lo <= hi) {
    const mid = lo + Math.floor((hi - lo) / 2)
    if (arr[mid] === target) return mid
    if (arr[mid] < target) lo = mid + 1
    else hi = mid - 1
  }
  return -1
}

console.log(search([1, 3, 5, 7, 9, 11], 7), search([1, 3, 5], 4))

// On the answer: smallest capacity to ship weights within D days
function shipWithinDays(w, days) {
  const daysNeeded = (cap) => { let d = 1, load = 0; for (const x of w) {
    if (load + x > cap) { d++; load = 0 } load += x } return d }
  let lo = Math.max(...w), hi = w.reduce((a, b) => a + b)
  while (lo < hi) { const mid = Math.floor((lo + hi) / 2); if
    (daysNeeded(mid) <= days) hi = mid; else lo = mid + 1 }
  return lo
}

console.log(shipWithinDays([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], 5)) // 15
```

8

Longest substring without repeating characters (sliding window).

Medium 🔥🔥 #sliding window #strings #coding

A **sliding window** keeps a range `[left, right]` that satisfies a condition. Expand `right` one step at a time; when the window breaks the rule, move `left` forward until it is valid again. Each index enters and leaves once, so it is $O(n)$ instead of checking every substring.

Here, store each character's last index in a map. When a repeat appears inside the window, jump `left` past its previous position.

- Time $O(n)$, space $O(k)$ for the alphabet

Use it for "longest/shortest subarray or substring such that...": max sum of size k, minimum window substring, longest with at most k distinct.

Example

```
function lengthOfLongestSubstring(s) {
  const last = new Map() // char -> last index seen
  let left = 0, best = 0
  for (let right = 0; right < s.length; right++) {
    const ch = s[right]
    if (last.has(ch) && last.get(ch) >= left) left = last.get(ch) + 1
    last.set(ch, right)
    best = Math.max(best, right - left + 1)
  }
  return best
}

console.log(lengthOfLongestSubstring('abcabcbb')) // 3 (abc)
console.log(lengthOfLongestSubstring('bbbbbb')) // 1
console.log(lengthOfLongestSubstring('pwwkew')) // 3 (wke)
console.log(lengthOfLongestSubstring('')) // 0
```

 **Likely follow-up:** Find the minimum window substring containing all characters of t.

9

What is the two-pointer technique? Give examples.

Medium



#two pointers #arrays #coding

Two indexes move through the data instead of nested loops, turning $O(n^2)$ into $O(n)$.

Patterns:

- **opposite ends** on sorted data: pair sum, container with most water, valid palindrome. If the sum is too small move **left** right, too big move **right** left.
- **same direction (read/write)**: remove duplicates in place, move zeroes
- **fast/slow**: cycle detection, middle of a linked list

The key is a reason why **moving one pointer can never skip the answer**; say it out loud in the interview.

Example below: container with most water. Always move the shorter wall, because keeping it can only give a smaller area.

Example

```
function maxArea(h) {
  let l = 0, r = h.length - 1, best = 0
  while (l < r) {
    best = Math.max(best, Math.min(h[l], h[r]) * (r - l))
    if (h[l] < h[r]) l++
    else r--
  }
  return best
}
console.log(maxArea([1, 8, 6, 2, 5, 4, 8, 3, 7])) // 49

function removeDuplicates(a) { // sorted array, in place
  let w = 0
  for (let i = 0; i < a.length; i++) if (i === 0 || a[i] !== a[i - 1])
    a[w++] = a[i]
  return a.slice(0, w)
}
console.log(removeDuplicates([1, 1, 2, 3, 3, 3, 4]))
```

10

Maximum subarray sum (Kadane's algorithm).

Medium 🔥🔥 #dp #arrays #coding

At each element decide: **extend the previous subarray or start fresh here.** `cur = max(x, cur + x)`; if the running sum went negative it can only hurt, so drop it. Track the best `cur` seen.

- Time `O(n)`, space `O(1)`

Initialise with the first element (not 0) so an all-negative array returns its largest element.

This is a tiny dynamic programming problem: `cur` is the best sum of a subarray ending at index `i`. Variations: maximum product subarray (track min and max), circular array.

Example

```
function maxSubArray(nums) {
  let cur = nums[0], best = nums[0]
  for (let i = 1; i < nums.length; i++) {
    cur = Math.max(nums[i], cur + nums[i])
    best = Math.max(best, cur)
  }
  return best
}
console.log(maxSubArray([-2, 1, -3, 4, -1, 2, 1, -5, 4])) // 6
([4,-1,2,1])
console.log(maxSubArray([-3, -1, -2])) // -1
console.log(maxSubArray([5, 4, -1, 7, 8])) // 23
```

11

Merge overlapping intervals.

Medium



#intervals #sorting #coding

Sort by start, then sweep. Keep the last merged interval; if the next one starts **at or before its end**, extend the end to the max of both ends; otherwise push it as a new interval.

- Time $O(n \log n)$ for the sort, space $O(n)$ for the output

Related interval problems use the same sort-then-sweep idea: insert interval, meeting rooms (can one person attend all?), minimum meeting rooms (sort starts and ends separately, or use a min-heap of end times), non-overlapping intervals.

Example

```
function merge(intervals) {
  const sorted = [...intervals].sort((a, b) => a[0] - b[0])
  const out = []
  for (const [s, e] of sorted) {
    const last = out[out.length - 1]
    if (last && s <= last[1]) last[1] = Math.max(last[1], e)
    else out.push([s, e])
  }
  return out
}

console.log(JSON.stringify(merge([[1, 3], [2, 6], [8, 10], [15, 18]])))
// [[1,6],[8,10],[15,18]]
console.log(JSON.stringify(merge([[1, 4], [4, 5]])))
// [[1,5]]
console.log(JSON.stringify(merge([[1, 4], [0, 4]])))
// [[0,4]]
```

? Likely follow-up: What is the minimum number of meeting rooms required?

12

Find the top K frequent elements. How does a heap help?

Medium 🔥🔥 #heap #hash map #coding

Count frequencies with a hash map, then keep a **min-heap of size k**: push each item, and if the heap grows past k, pop the smallest. What remains are the k largest.

- Time $O(n \log k)$, space $O(n)$ for the counts
- Alternative: **bucket sort** by frequency (index = count) for $O(n)$

A heap gives $O(\log n)$ insert and remove of the min (or max) and $O(1)$ peek.

JavaScript has no built-in heap, so be ready to write a small one (below) or explain that you would.

Use heaps for: top k, k closest points, merging k sorted lists, running median (two heaps), Dijkstra.

Example

```
class MinHeap {
  constructor(key) { this.a = []; this.key = key }
  get size() { return this.a.length }
  push(x) { const a = this.a; a.push(x); let i = a.length - 1
    while (i > 0) { const p = (i - 1) >> 1; if (this.key(a[p]) <=
this.key(a[i])) break; [a[p], a[i]] = [a[i], a[p]]; i = p } }
  pop() { const a = this.a, top = a[0], last = a.pop()
    if (a.length) { a[0] = last; let i = 0
      while (true) { let m = i; const l = 2 * i + 1, r = l + 1
        if (l < a.length && this.key(a[l]) < this.key(a[m])) m = l
        if (r < a.length && this.key(a[r]) < this.key(a[m])) m = r
        if (m === i) break; [a[m], a[i]] = [a[i], a[m]]; i = m } }
      return top }
}

function topKFrequent(nums, k) {
  const count = new Map()
  for (const n of nums) count.set(n, (count.get(n) ?? 0) + 1)
  const heap = new MinHeap(([, c]) => c)
  for (const entry of count) { heap.push(entry); if (heap.size > k)
heap.pop() }
  return heap.a.map(([n]) => n)
}

console.log(topKFrequent([1, 1, 1, 2, 2, 3], 2)) // [2, 1] in some
order
console.log(topKFrequent([4, 4, 5, 6, 6, 6, 7], 1)) // [6]
```

13

Find the kth largest element in an array.

Medium 🔥🔥 #heap #quickselect #coding

Three answers, from simplest to best:

- **Sort** descending and take index $k - 1$: $O(n \log n)$
- **Min-heap of size k**: the top is the kth largest, $O(n \log k)$, good for streams
- **Quickselect**: partition like quicksort but recurse into only one side. Average $O(n)$, worst $O(n^2)$ (avoid with a random pivot), $O(1)$ extra space

Mention all three and implement quickselect or the heap depending on what the interviewer wants. For the kth largest, look for index $n - k$ in ascending order.

Example

```
function findKthLargest(nums, k) {
  const a = [...nums], target = a.length - k
  let lo = 0, hi = a.length - 1
  while (true) {
    const r = lo + Math.floor(Math.random() * (hi - lo + 1))
    ;[a[r], a[hi]] = [a[hi], a[r]]
    let p = lo // Lomuto partition
    for (let i = lo; i < hi; i++) if (a[i] < a[hi]) { [a[i], a[p]] =
[a[p], a[i]]; p++ }
    ;[a[p], a[hi]] = [a[hi], a[p]]
    if (p === target) return a[p]
    if (p < target) lo = p + 1
    else hi = p - 1
  }
}

console.log(findKthLargest([3, 2, 1, 5, 6, 4], 2)) // 5
console.log(findKthLargest([3, 2, 3, 1, 2, 4, 5, 5, 6], 4)) // 4
```

14

BFS vs DFS: how do they differ and when do you use each?

Medium



#graphs #bfs #dfs

BFS explores level by level using a **queue**. It finds the **shortest path in an unweighted graph** and suits "minimum steps" problems (word ladder, rotting oranges, nearest exit). Memory grows with the widest level.

DFS goes as deep as possible using **recursion or a stack**, then backtracks. Natural for exhaustive search, connected components, cycle detection, topological sort, path existence and backtracking (permutations, N-Queens). Memory grows with the depth.

Both are $O(V + E)$ time and need a **visited** set on graphs (not on trees). Deep recursion can overflow the stack; switch to an explicit stack.

Example

```
// Number of islands (DFS) and shortest path in a grid (BFS)
const grid = ['11000', '11010', '00011'].map((r) => [...r])
function islands(g) {
  const seen = new Set(), R = g.length, C = g[0].length
  const dfs = (r, c) => {
    if (r < 0 || c < 0 || r >= R || c >= C || g[r][c] !== '1' ||
    seen.has(r * C + c)) return
    seen.add(r * C + c); dfs(r + 1, c); dfs(r - 1, c); dfs(r, c + 1);
    dfs(r, c - 1)
  }
  let n = 0
  for (let r = 0; r < R; r++) for (let c = 0; c < C; c++) if (g[r][c] ===
  '1' && !seen.has(r * C + c)) { n++; dfs(r, c) }
  return n
}
function shortest(g, [sr, sc], [tr, tc]) {
  const q = [[sr, sc, 0]], seen = new Set([sr + ',' + sc])
  for (const [r, c, d] of q) { // array used as a queue
    if (r === tr && c === tc) return d
    for (const [nr, nc] of [[r + 1, c], [r - 1, c], [r, c + 1], [r, c -
    1]])
      if (g[nr]?.[nc] === '0' && !seen.has(nr + ',' + nc)) { seen.add(nr
      + ',' + nc); q.push([nr, nc, d + 1]) }
  }
  return -1
}
console.log('islands:', islands(grid)) // 2
console.log('steps:', shortest(grid, [2, 0], [0, 4])) // 6
```

15

Explain tree traversals: preorder, inorder, postorder, level order.

Easy



#trees #coding

Depth-first orders differ in **when you visit the node** relative to its children:

- **Preorder** (node, left, right): copy or serialise a tree
- **Inorder** (left, node, right): on a BST it gives **values in sorted order**
- **Postorder** (left, right, node): delete a tree, compute heights or sizes bottom-up

Level order (BFS with a queue) visits row by row: print levels, right side view, min depth.

All are $O(n)$ time. DFS uses $O(h)$ stack space (h = height, $O(n)$ if skewed); level order uses $O(w)$ for the widest level. Be ready to write inorder iteratively with an explicit stack.

Example

```
const t = (val, left = null, right = null) => ({ val, left, right })
//      4
//     2  6
//    1 3 5 7
const root = t(4, t(2, t(1), t(3)), t(6, t(5), t(7)))
const pre = (n) => (n ? [n.val, ...pre(n.left), ...pre(n.right)] : [])
const post = (n) => (n ? [...post(n.left), ...post(n.right), n.val] : [])
function inorder(n) { // iterative
  const out = [], stack = []
  while (n || stack.length) {
    while (n) { stack.push(n); n = n.left }
    n = stack.pop(); out.push(n.val); n = n.right
  }
  return out
}
function levels(r) {
  const out = []; let q = r ? [r] : []
  while (q.length) { out.push(q.map((n) => n.val)); q = q.flatMap((n) => [n.left, n.right].filter(Boolean)) }
  return out
}
console.log('pre ', pre(root).join(' '))
console.log('in  ', inorder(root).join(' '))
console.log('post', post(root).join(' '))
console.log('levels', JSON.stringify(levels(root)))
```

16

Find the maximum depth of a binary tree.

Easy



#trees #recursion #coding

Recursive definition: an empty tree has depth 0; otherwise depth is $1 + \max(\text{depth}(\text{left}), \text{depth}(\text{right}))$. This is a postorder computation: children first, then the node.

- Time $O(n)$, space $O(h)$ for recursion

Iterative alternative: level-order BFS and count the levels.

The same "ask children, combine at the parent" shape solves many tree questions: is the tree balanced, diameter of a tree (longest path through each node is left height + right height), path sum, invert a tree.

Example

```
const t = (val, left = null, right = null) => ({ val, left, right })
const maxDepth = (n) => (n ? 1 + Math.max(maxDepth(n.left),
maxDepth(n.right)) : 0)

function diameter(root) {
  let best = 0
  const h = (n) => {
    if (!n) return 0
    const l = h(n.left), r = h(n.right)
    best = Math.max(best, l + r) // longest path through n (edges)
    return 1 + Math.max(l, r)
  }
  h(root)
  return best
}
const root = t(3, t(9), t(20, t(15), t(7, t(1))))
console.log(maxDepth(root), maxDepth(null)) // 4 0
console.log(diameter(root)) // 4
```

17

Check whether a binary tree is a valid binary search tree.

Medium 🔥🔥 #trees #bst #coding

The trap: checking only `left.val < node.val < right.val` is not enough. Every node in the left **subtree** must be smaller, not just the direct child.

Correct approach: pass down an allowed range `(min, max)`. Going left, the **node's value becomes the new max**; going right, it becomes the new min. Every node must be strictly within its range.

- Time `O(n)`, space `O(h)`

Alternative: an inorder traversal must be strictly increasing.

Example

```
const t = (val, left = null, right = null) => ({ val, left, right })
function isValidBST(n, min = -Infinity, max = Infinity) {
  if (!n) return true
  if (n.val <= min || n.val >= max) return false
  return isValidBST(n.left, min, n.val) && isValidBST(n.right, n.val,
max)
}
console.log(isValidBST(t(2, t(1), t(3)))) // true
console.log(isValidBST(t(5, t(1), t(4, t(3), t(6))))) // false
console.log(isValidBST(t(5, t(4), t(6, t(3), t(7))))) // false: 3 is
in 5's right subtree
```

18

Lowest common ancestor in a BST (and in a plain binary tree).

Medium 🔥 #trees #bst #coding

In a BST, use the ordering: if both values are smaller than the node, go left; if both are larger, go right; otherwise the paths **split here, so this node is the LCA** (this includes when it equals one of them).

- Time $O(h)$, space $O(1)$ iteratively

In a general binary tree, recurse: return the node if it is **p** or **q**; ask both children; if both return non-null, the current node is the LCA, otherwise pass up the non-null side.

- Time $O(n)$, space $O(h)$

Example

```
const t = (val, left = null, right = null) => ({ val, left, right })
//      6
//    2   8
//  0  4  7  9
//    3 5
const root = t(6, t(2, t(0), t(4, t(3), t(5))), t(8, t(7), t(9)))
function lcaBST(n, p, q) {
  while (n) {
    if (p < n.val && q < n.val) n = n.left
    else if (p > n.val && q > n.val) n = n.right
    else return n.val
  }
  return null
}
function lca(n, p, q) { // any binary tree
  if (!n || n.val === p || n.val === q) return n
  const l = lca(n.left, p, q), r = lca(n.right, p, q)
  return l && r ? n : l || r
}
console.log(lcaBST(root, 2, 8), lcaBST(root, 2, 4), lcaBST(root, 3, 5))
// 6 2 4
console.log(lca(root, 7, 9).val, lca(root, 0, 5).val)
// 8 2
```

19

Recursion vs iteration: trade-offs?

Easy



#recursion

Any recursion can be written as iteration with an explicit stack, and vice versa.

Recursion is natural for trees, graphs, divide and conquer and backtracking: short, close to the definition. Each call adds a stack frame, so it uses $O(\text{depth})$ memory and **can overflow the call stack** (around 10^4 frames in JS engines, and JS has no reliable tail-call optimisation).

Iteration is usually faster with no stack limit, but tree/graph code with an explicit stack is wordier.

Every recursive solution needs a **base case** and progress toward it. If subproblems repeat (naive Fibonacci is $O(2^n)$), add memoisation.

Example

```
const sumRec = (n) => (n === 0 ? 0 : n + sumRec(n - 1))
function sumIter(n) { let s = 0; for (let i = 1; i <= n; i++) s += i;
return s }
console.log(sumRec(100), sumIter(100))
try { sumRec(1e6) } catch (e) { console.log('recursion:', e.name) } //
RangeError: stack overflow
console.log('iteration:', sumIter(1e6))
```

20

Dynamic programming: memoisation vs tabulation (climbing stairs, coin change).

Medium 🔥🔥 #dp #coding

DP applies when a problem has **overlapping subproblems** and **optimal substructure**. Define the state and the recurrence first.

- **Memoisation (top-down)**: write the natural recursion and cache results by state. Easy to derive; only computes needed states; recursion overhead.
- **Tabulation (bottom-up)**: fill a table from the base cases up in a loop. No recursion limit, and often you can **keep only the last few values** to save space.

Climbing stairs (1 or 2 steps): $\text{ways}(n) = \text{ways}(n-1) + \text{ways}(n-2)$; $O(n)$ time, $O(1)$ space.

Coin change (fewest coins for amount): $\text{dp}[a] = \min(\text{dp}[a - c] + 1)$ over coins; $O(\text{amount} * \text{coins})$ time, $O(\text{amount})$ space. Greedy fails for coins like [1, 3, 4] and amount 6.

Example

```
function climbMemo(n, memo = new Map()) { // top-down
  if (n <= 2) return n
  if (!memo.has(n)) memo.set(n, climbMemo(n - 1, memo) + climbMemo(n - 2, memo))
  return memo.get(n)
}
function climbTab(n) { // bottom-up, O(1)
  space
  let a = 1, b = 1
  for (let i = 2; i <= n; i++) [a, b] = [b, a + b]
  return b
}
function coinChange(coins, amount) {
  const dp = new Array(amount + 1).fill(Infinity)
  dp[0] = 0
  for (let a = 1; a <= amount; a++)
    for (const c of coins) if (c <= a) dp[a] = Math.min(dp[a], dp[a - c] + 1)
  return dp[amount] === Infinity ? -1 : dp[amount]
}
console.log(climbMemo(10), climbTab(10), climbTab(45)) // 89 89 1836311903
console.log(coinChange([1, 3, 4], 6)) // 2 (3+3), greedy would give 3
console.log(coinChange([2], 3)) // -1
```

? **Likely follow-up:** Count the number of ways to make the amount instead of the fewest coins.

21

Design and implement an LRU cache with O(1) get and put.

Medium



#design #hash map #linked list

Classic design: a **hash map** for **O(1)** lookup plus a **doubly linked list** ordered by recency. On **get**, move the node to the front; on **put**, insert at the front and, if over capacity, **evict the node at the tail**. Map and list together give **O(1)** for everything.

In JavaScript a **Map** already keeps insertion order, so a short version works: on access, delete and re-insert the key (now newest); evict **map.keys().next().value** (oldest). Mention that this relies on Map ordering and that in other languages you would build the linked list (or use **LinkedHashMap** in Java).

Example

```
class LRUCache {
  constructor(capacity) { this.cap = capacity; this.map = new Map() }
  get(key) {
    if (!this.map.has(key)) return -1
    const v = this.map.get(key)
    this.map.delete(key); this.map.set(key, v) // mark most recent
    return v
  }
  put(key, value) {
    this.map.delete(key)
    this.map.set(key, value)
    if (this.map.size > this.cap)
      this.map.delete(this.map.keys().next().value) // oldest
  }
}

const c = new LRUCache(2)
c.put(1, 'a'); c.put(2, 'b')
console.log(c.get(1)) // a (1 is now most recent)
c.put(3, 'c') // evicts 2
console.log(c.get(2), c.get(3), [...c.map.keys()]) // -1 c [1, 3]
```

 **Likely follow-up:** Implement it with an explicit doubly linked list.

22

Compare quicksort, mergesort and heapsort. What is a stable sort?

Medium 🔥🔥 #sorting

- **Quicksort:** average $O(n \log n)$, worst $O(n^2)$ (bad pivots; randomise), $O(\log n)$ stack space, in place, **not stable**. Fastest in practice thanks to cache locality.
- **Mergesort:** always $O(n \log n)$, needs $O(n)$ extra space, **stable**. Great for linked lists and external sorting of huge files.
- **Heapsort:** always $O(n \log n)$, $O(1)$ space, not stable, slower in practice.
- Simple sorts (insertion) are $O(n^2)$ but fast on tiny or nearly sorted arrays.

A **stable** sort keeps equal elements in their original order, which matters when sorting by one key after another. JS `Array.prototype.sort` is stable (TimSort, a merge/insertion hybrid). Comparison sorts cannot beat $O(n \log n)$; counting/radix sort can for small integer ranges.

Example

```
function mergeSort(a) {
  if (a.length <= 1) return a
  const mid = a.length >> 1
  const l = mergeSort(a.slice(0, mid)), r = mergeSort(a.slice(mid))
  const out = []
  let i = 0, j = 0
  while (i < l.length && j < r.length) out.push(l[i].age <= r[j].age ?
l[i++] : r[j++]) // <= keeps it stable
  return out.concat(l.slice(i), r.slice(j))
}

const people = [{ n: 'Asha', age: 30 }, { n: 'Ben', age: 25 }, { n:
'Cai', age: 30 }, { n: 'Dev', age: 25 }]
console.log(mergeSort(people).map((p) => p.n + p.age).join(' ')) // Ben25
Dev25 Asha30 Cai30
```

? Likely follow-up: Why is quicksort usually faster than mergesort in practice?

23

Group anagrams together.

Medium 🔥 #hash map #strings #coding

Anagrams share the same letters, so give each word a **canonical key** and group by it in a hash map.

- **Sorted letters** as key ('eat' becomes 'aet'): $O(n * k \log k)$ for n words of length k
- **Letter-count signature** (26 counts joined): $O(n * k)$, better for long words

Space $O(n * k)$ for the map.

The same canonical-key idea answers "valid anagram" (compare counts) and "find all anagrams of p in s" (sliding window with counts).

Example

```
function groupAnagrams(words) {
  const groups = new Map()
  for (const w of words) {
    const count = new Array(26).fill(0)
    for (const ch of w) count[ch.charCodeAt(0) - 97]++
    const key = count.join(',')
    if (!groups.has(key)) groups.set(key, [])
    groups.get(key).push(w)
  }
  return [...groups.values()]
}
console.log(JSON.stringify(groupAnagrams(['eat', 'tea', 'tan', 'ate', 'nat', 'bat'])))
// [["eat","tea","ate"],["tan","nat"],["bat"]]
```

24

What is a trie and when would you use one?

Medium



#trie #strings #coding

A **trie** (prefix tree) stores strings character by character: each node has children keyed by letter and an **end** flag. Words with a common prefix share the same path.

- insert and search are **$O(L)$** in the word length, independent of how many words are stored
- **`startsWith(prefix)`** is also **$O(L)$** , which a hash set cannot do

Uses: autocomplete and search suggestions, spell checkers, word games (Word Search II: put all words in a trie and DFS the board once), IP routing (longest prefix match).

Cost: memory, since every node holds a map of children.

Example

```
class Trie {
  constructor() { this.root = { children: new Map(), end: false } }
  insert(word) {
    let n = this.root
    for (const ch of word) {
      if (!n.children.has(ch)) n.children.set(ch, { children: new Map(),
end: false })
      n = n.children.get(ch)
    }
    n.end = true
  }
  #walk(s) { let n = this.root; for (const ch of s) { n =
n.children.get(ch); if (!n) return null } return n }
  search(word) { return this.#walk(word)?.end === true }
  startsWith(prefix) { return this.#walk(prefix) !== null }
}
const trie = new Trie()
for (const w of ['car', 'card', 'care', 'dog']) trie.insert(w)
console.log(trie.search('car'), trie.search('ca'), trie.startsWith('ca'),
trie.startsWith('x'))
// true false true false
```

25

Detect a cycle in a directed graph and produce a topological order (course schedule).

Medium



#graphs #topological sort #coding

A **topological order** lists nodes so every edge $u \rightarrow v$ has u before v (build order, course prerequisites). It exists only if the graph is a **DAG** (no cycle).

Kahn's algorithm (BFS): count incoming edges; queue all nodes with in-degree 0; pop a node, append it to the order, and decrement its neighbours, queueing any that reach 0. If the order has fewer nodes than the graph, there is a cycle.

DFS alternative: three colours (unvisited, in progress, done); meeting an "in progress" node means a back edge, i.e. a cycle. The reverse postorder is a topological order.

Time $O(V + E)$, space $O(V + E)$.

Example

```
function courseOrder(n, prereqs) { // [course, needs]
  const adj = Array.from({ length: n }, () => [])
  const indeg = new Array(n).fill(0)
  for (const [c, p] of prereqs) { adj[p].push(c); indeg[c]++ }
  const queue = []
  for (let i = 0; i < n; i++) if (indeg[i] === 0) queue.push(i)
  const order = []
  for (const u of queue) { // queue grows while we
iterate
    order.push(u)
    for (const v of adj[u]) if (--indeg[v] === 0) queue.push(v)
  }
  return order.length === n ? order : null // null: cycle
}
console.log(courseOrder(4, [[1, 0], [2, 0], [3, 1], [3, 2]])) // [0, 1,
2, 3]
console.log(courseOrder(2, [[0, 1], [1, 0]])) // null
```

26

How does Dijkstra's algorithm work?

Hard 🔥🔥 #graphs #shortest path #coding

Dijkstra finds the **shortest paths from one source** in a graph with **non-negative edge weights**.

Keep **dist** (Infinity except source = 0) and a **min-priority queue** of **(distance, node)**. Repeatedly pop the closest unfinished node; its distance is now final. **Relax** each edge: if **dist[u] + w < dist[v]**, update **dist[v]** and push it. Skip stale queue entries.

- Time **$O((V + E) \log V)$** with a binary heap, space **$O(V + E)$**

Negative weights break it (use **Bellman-Ford**). Unweighted graphs only need BFS. For grids with a target, **A*** adds a heuristic.

Example

```
function dijkstra(graph, src) {
  const dist = Object.fromEntries(Object.keys(graph).map((k) => [k,
  Infinity]))
  dist[src] = 0
  const pq = [[0, src]] // sorted array as a simple PQ
  (use a heap for big graphs)
  while (pq.length) {
    const [d, u] = pq.shift()
    if (d > dist[u]) continue // stale entry
    for (const [v, w] of graph[u]) {
      if (d + w < dist[v]) {
        dist[v] = d + w
        pq.push([dist[v], v])
        pq.sort((a, b) => a[0] - b[0])
      }
    }
  }
  return dist
}

const graph = { A: [['B', 4], ['C', 1]], B: [['D', 1]], C: [['B', 2],
['D', 5]], D: [] }
console.log(dijkstra(graph, 'A')) // { A: 0, B: 3, C: 1, D: 4 }
```

? Likely follow-up: Why does Dijkstra fail with negative edge weights?

27

Sort an array of 0s, 1s and 2s in one pass (Dutch national flag).

Medium 🔥 #two pointers #sorting #coding

Three pointers split the array into regions: `[0, low)` holds 0s, `[low, mid)` 1s, `(high, end]` 2s, and `[mid, high]` is unknown.

- `a[mid] === 0`: swap with `low`, advance both
- `a[mid] === 1`: just advance `mid`
- `a[mid] === 2`: swap with `high`, decrement `high`, but **do not advance `mid`**, because the swapped-in value is still unchecked

Time `O(n)` in a single pass, space `O(1)`. Counting sort (count, then overwrite) is also `O(n)` but takes two passes. This partition is also the three-way partition used in quicksort for many duplicates.

Example

```
function sortColors(a) {
  let low = 0, mid = 0, high = a.length - 1
  while (mid <= high) {
    if (a[mid] === 0) { [a[low], a[mid]] = [a[mid], a[low]]; low++; mid++; }
    else if (a[mid] === 1) mid++;
    else { [a[mid], a[high]] = [a[high], a[mid]]; high-- }
  }
  return a
}

console.log(sortColors([2, 0, 2, 1, 1, 0])) // [0, 0, 1, 1, 2, 2]
console.log(sortColors([2, 0, 1]))         // [0, 1, 2]
console.log(sortColors([1, 1, 1]))
```

28

How do you approach a coding problem you have never seen before?

Easy



#strategy

Interviewers grade the process as much as the code:

- **Clarify:** restate the problem, ask about input size, ranges, duplicates, negatives, empty input, expected output format
- **Examples:** work a small case by hand, plus an edge case
- **Brute force first:** say it and its complexity, so you have a baseline
- **Optimise** by matching patterns: sorted input suggests binary search or two pointers; "subarray/substring" suggests sliding window or prefix sums; "top k" a heap; "number of ways/min cost" DP; relations a graph; lookups a hash map
- **Use the constraints:** n up to 10^5 needs $O(n \log n)$; n up to 20 allows $O(2^n)$
- **Code** cleanly with good names, **talking through your reasoning**
- **Test:** dry-run your examples and edge cases, then state time and space complexity