

Roblox — Senior SWE, Safety Foundation

45-minute coding interview · overnight cram sheet

What this is. A ranked bet on what you'll actually be asked, built from company-tagged LeetCode frequency data for Roblox (all-time + last-6-months buckets), published Roblox interview guides, and first-hand candidate reports — then filtered through what the **Safety Foundation** team actually builds.

The one-line thesis: Roblox's recent coding rounds are dominated by “**design a small stateful object over a stream of timestamped events**” and “**simulate this grid/board**”. Safety Foundation builds rate limiters, text filters and moderation workflow engines. Those two facts point at the **same** five problems.

Your interview, specifically

Safety Foundation is the infra backbone for all moderation at Roblox: every piece of content reviewed and every moderation action flows through it. The JD calls out **high-scale event-driven workflow systems, AI-powered text filters, abuse/spam detection, and auditability**. Roblox's own guides say coding problems are LeetCode-medium and are often **disguised in a gaming or product context** rather than stated as a raw LeetCode problem.

A 45-minute slot with a single senior engineer almost certainly means **one medium problem plus 2–3 follow-ups** — not two problems. The follow-ups are where the senior signal is scored. Budget accordingly.

Minute	What should be happening
0–4	Intros, brief context on the team. Be warm, be brief.
4–8	Clarify. Input size, duplicates, ordering, single-threaded?, unicode? Ask before coding.
8–12	State the approach + complexity out loud. Get a nod before you type. This is the #1 thing juniors skip.
12–33	Code it. Narrate. Clean names.
33–40	Dry-run a tiny example by hand. Then edge cases.
40–45	Follow-ups + your questions for them.

The ranked bets

#	Pattern	Confidence
1	Rate limiter / hit counter — stateful object over a timestamped stream	Very high
2	Trie / prefix matching — chat filter, banned words, autocomplete	High
3	Grid / board simulation — Candy Crush, gravity, BFS flood fill	High
4	Sliding window + hash map over a sequence	Medium-high
5	Heap / greedy scheduling with cooldown; interval merging	Medium
6	Graph / topological sort (workflow dependencies)	Medium
7	Hash-map aggregation (likely as the warm-up)	Medium

Why these — the evidence

- **Last-6-months** Roblox-tagged LeetCode frequency, top of list: *Logger Rate Limiter* (100), *Candy Crush* (95.7), *Design Search Autocomplete System* (83.8), *Design Hit Counter* (74.3), *Two Sum* (74.3), *Maximize Distance to Closest Person* (74.3).
- **All-time** Roblox-tagged top: *Candy Crush* (100), *Reorganize String* (94.7), *Text Justification* (89.4), *Task Scheduler* (85.6), *Course Schedule II* (83.4), *Design Hit Counter* (76.7). A second dataset ranks *Subdomain Visit Count* #1 by raw frequency.
- A 2026 Roblox process breakdown lists the low-level-design staples verbatim as “**rate limiters, resource loaders, and hit counters**”, and names **sliding-window rate limiter** as a reported question.
- Crowd-sourced question banks carry **Roblox-tagged** entries “implement queue-based rate limiter with multi-key limits” and “design sliding-window rate limiter with multi-keys”.
- An Oct-2025 candidate report: a Roblox round was “**heavily centered around graph and trie questions, which were very implementation-heavy**”.
- A published Roblox question list includes “**design an in-game chat filter that detects and redacts inappropriate language**” and “find the most frequent event within any given hour”.

If you only have one hour tonight: type out #1 (rate limiter, all three variants) and #2 (trie + censor). They cover the highest-probability outcome and they are the two most on-theme for a Safety team. Skim the rest.

1. Rate limiter / hit counter

Confidence: very high. This is the single most likely thing you will see. It is #1 in the recent frequency data, it is named explicitly in Roblox process guides, and it is literally what an anti-spam/anti-abuse platform team builds. Expect it phrased as a product story: “*we don't want to alert on the same abuse signal more than once every N seconds*” or “*cap each user at K chat messages per minute*”.

1a. The warm-up form: Logger Rate Limiter (LC 359)

Print a message only if it hasn't been printed in the last 10 seconds. Timestamps arrive in non-decreasing order.

```
class Logger:
    def __init__(self):
        self.last_printed = {}           # msg -> time we last printed it

    def shouldPrintMessage(self, t: int, msg: str) -> bool:
        prev = self.last_printed.get(msg)
        if prev is None or t - prev >= 10:
            self.last_printed[msg] = t
            return True
        return False
```

O(1) per call. The trap: only update the timestamp when you actually *print*. If you update on every call, a spammer sending every 9s is silenced forever — wrong behaviour.

1b. The most likely form: Design Hit Counter (LC 362)

Count hits in the trailing 5-minute (300s) window.

```
from collections import deque

class HitCounter:
    def __init__(self):
        self.q = deque()                # timestamps, non-decreasing

    def hit(self, t: int) -> None:
        self.q.append(t)

    def getHits(self, t: int) -> int:
        while self.q and self.q[0] <= t - 300:
            self.q.popleft()
        return len(self.q)
```

Amortised **O(1)**: every timestamp is pushed once and popped once.

Follow-up they will absolutely ask: “what if there are thousands of hits per second?”

Don't store every timestamp — **bucket** them. Fixed ring of 300 slots:

```
class HitCounter:
    def __init__(self, window=300):
```

```

self.w = window
self.times = [0] * window      # which second this slot holds
self.counts = [0] * window     # hits during that second

def hit(self, t):
    i = t % self.w
    if self.times[i] != t:      # slot is stale -> reset it
        self.times[i], self.counts[i] = t, 0
    self.counts[i] += 1

def getHits(self, t):
    return sum(c for tm, c in zip(self.times, self.counts)
               if t - tm < self.w)

```

O(1) memory regardless of traffic. Saying “the deque is unbounded in memory, so let me bucket it” *before* they ask is a strong senior signal.

1c. The Safety-flavoured form: per-client sliding window

This is the version explicitly reported as a Roblox question (“multi-key” = per user/IP/experience). If they open with 1a or 1b, this is where the follow-ups go.

```
from collections import defaultdict, deque
```

```

class RateLimiter:
    """Allow at most `limit` actions per `window` seconds, per key."""
    def __init__(self, limit: int, window: int):
        self.limit, self.window = limit, window
        self.hits = defaultdict(deque)

    def allow(self, key: str, t: int) -> bool:
        q = self.hits[key]
        while q and q[0] <= t - self.window:
            q.popleft()
        if len(q) < self.limit:
            q.append(t)
            return True
        return False                    # rejected calls do NOT count

```

The three follow-ups to have ready

- Unbounded memory across keys.** `self.hits` grows forever as new users appear. Fix: evict on read, plus a periodic sweep, or an LRU cap (`OrderedDict + move_to_end + popitem(last=False)`). Say this unprompted.
- Different limits per key / tiered limits.** Store (`limit`, `window`) per key or per tier; or chain several limiters (10/sec AND 100/min AND 1000/hour) and require all to pass. Mention that you check cheapest-first and only commit the timestamp if every limiter allows — otherwise you leak quota.
- Distributed / multi-server.** In-process state doesn't work behind a load balancer. Options: sticky routing by key, or centralise in Redis (`INCR + EXPIRE`, or a sorted set per key), accepting a small race. Note the trade-off: exact counting costs a round trip; approximate local counting is faster but over-admits.

Know the alternative: token bucket

Mention it by name even if you don't code it. It's $O(1)$ memory per key, allows controlled bursts, and is what real infra uses.

```
class TokenBucket:
    def __init__(self, rate, capacity):      # rate = tokens/second
        self.rate, self.cap = rate, capacity
        self.tokens = float(capacity)
        self.last = 0.0

    def allow(self, now: float, n: int = 1) -> bool:
        self.tokens = min(self.cap,
                          self.tokens + (now - self.last) * self.rate)
        self.last = now
        if self.tokens >= n:
            self.tokens -= n
            return True
        return False
```

Safety-team bonus points. Drop these in naturally: use a **monotonic clock**, not wall clock, so NTP jumps can't unlock the limiter. **Fail open or fail closed?** — for spam throttling you usually fail open (availability); for child-safety enforcement you fail closed. And every reject should be **auditable**, because the JD is explicit about auditability.

Sibling problems in the same family

- *Number of Recent Calls* (LC 933) — identical deque, 3000ms window.
- *Logger Rate Limiter* (LC 359) — 1a above.
- *Design Hit Counter* (LC 362) — 1b above.
- “Most frequent event in any given hour” — deque window + Counter, tracking the max as you slide.

2. Trie / prefix matching

Confidence: high. *Design Search Autocomplete System* is #4 in the recent data; *Implement Trie*, *Word Search II*, and two longest-common-prefix problems also appear. An Oct-2025 candidate reported a round “heavily centered around graph and trie questions.” And the Safety JD names **AI-powered text filters** — Roblox’s chat filter is the team’s most famous artifact. A published Roblox question list literally contains “design an in-game chat filter that detects and redacts inappropriate language.”

The base you must be able to type from memory

```
class TrieNode:
    __slots__ = ("children", "is_word", "word")
    def __init__(self):
        self.children = {}           # char -> TrieNode
        self.is_word = False
        self.word = None            # used by Word Search II below

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word: str) -> None:
        node = self.root
        for ch in word:
            node = node.children.setdefault(ch, TrieNode())
        node.is_word = True

    def _walk(self, s):
        node = self.root
        for ch in s:
            if ch not in node.children:
                return None
            node = node.children[ch]
        return node

    def search(self, word: str) -> bool:
        n = self._walk(word)
        return n is not None and n.is_word

    def startsWith(self, prefix: str) -> bool:
        return self._walk(prefix) is not None
```

setdefault is the trick that makes insert four lines. Insert/search are **$O(\text{len}(\text{word}))$** .

The Safety version: redact banned words from chat

Scan every start position, walk the trie as far as it goes, take the **longest** match. This is the poor-man’s Aho-Corasick and is what they want in 45 minutes.

```
def censor(text: str, banned: list[str], mask: str = "**") -> str:
    trie = Trie()
    for w in banned:
```

```

    if w:
        trie.insert(w.lower())

low = text.lower()
out = list(text)
i, n = 0, len(low)
while i < n:
    node, j, end = trie.root, i, -1
    while j < n and low[j] in node.children:
        node = node.children[low[j]]
        j += 1
    if node.is_word:
        end = j
    if end != -1:
        for k in range(i, end):
            out[k] = mask
        i = end
    else:
        i += 1
    # remember longest match so far
    # skip past the redacted span
return "".join(out)

```

$O(n \cdot L)$ where L is the longest banned word; **$O(\text{total chars in dictionary})$** memory.

Follow-ups to pre-load

- **“Users write f**k, f-u-c-k, fuuuck.”** Normalise before matching: lowercase, strip non-alphanumerics, map leetspeak (0→o, 1→i, 3→e, \$→s), collapse runs of repeated letters. Crucially, keep an index map from normalised position back to original position so you can redact the right characters.
- **“Scunthorpe problem.”** Substring matching produces false positives inside innocent words. Fix: require word boundaries, or maintain an allow-list trie checked first. Naming this problem out loud is a real signal for a Safety role.
- **“Millions of terms, updated live.”** Mention Aho-Corasick (adds failure links, gives true $O(n + \text{matches})$ in one pass) and that the trie can be rebuilt off-thread and hot-swapped by pointer to avoid locking readers.
- **“Top-3 completions for a prefix”** (autocomplete): store a small sorted list of best (count, string) at each node, or walk the subtree and use `heapq.nsmallest(3, ...)` with key `(-count, word)`.

Word Search II — trie + grid DFS, if they combine 2 and 3

```

def findWords(board, words):
    root = TrieNode()
    for w in words:
        node = root
        for ch in w:
            node = node.children.setdefault(ch, TrieNode())
        node.word = w

    R, C, found = len(board), len(board[0]), []

    def dfs(r, c, node):
        ch = board[r][c]

```

```

nxt = node.children.get(ch)
if not nxt:
    return
w = nxt.word
if w:
    found.append(w)
    nxt.word = None          # de-dupe
    board[r][c] = "#"        # mark visited
    for dr, dc in ((1,0), (-1,0), (0,1), (0,-1)):
        nr, nc = r + dr, c + dc
        if 0 <= nr < R and 0 <= nc < C and board[nr][nc] != "#":
            dfs(nr, nc, nxt)
    board[r][c] = ch          # restore

for r in range(R):
    for c in range(C):
        dfs(r, c, root)
return found

```

3. Grid / board simulation

Confidence: high. *Candy Crush* is **#1 all-time** and **#2** in the last six months for Roblox. Add *Rotating the Box*, *Rotate Image*, *Spiral Matrix*, *Image Smoother*, *Rotting Oranges*, and Roblox's own guides describing problems “disguised in a gaming or simulation context”. This is Roblox's house style. It is rarely clever — it rewards **not making off-by-one errors under time pressure**.

Candy Crush (LC 723) — the archetype

Two phases, repeated to stability: **mark** all runs of 3+ (using negatives so you don't disturb detection mid-pass), then apply **gravity** per column.

```
def candyCrush(board):
    R, C = len(board), len(board[0])
    found = False

    for r in range(R):
        # mark horizontal runs
        for c in range(C - 2):
            v = abs(board[r][c])
            if v and v == abs(board[r][c+1]) == abs(board[r][c+2]):
                board[r][c] = board[r][c+1] = board[r][c+2] = -v
                found = True

    for r in range(R - 2):
        # mark vertical runs
        for c in range(C):
            v = abs(board[r][c])
            if v and v == abs(board[r+1][c]) == abs(board[r+2][c]):
                board[r][c] = board[r+1][c] = board[r+2][c] = -v
                found = True

    if not found:
        return board

    for c in range(C):
        # gravity: write pointer
        w = R - 1
        for r in range(R - 1, -1, -1):
            if board[r][c] > 0:
                board[w][c] = board[r][c]
                w -= 1
        for r in range(w, -1, -1):
            board[r][c] = 0
    return candyCrush(board) # repeat until stable
```

Why negatives? You cannot delete as you scan — a cell can belong to both a horizontal and a vertical run, and deleting early breaks the other detection. Marking with the sign bit lets you keep the value for comparison (abs) while recording “doomed”. Explain this out loud; it's the whole insight.

Bottom-up write pointer is the gravity idiom. Same shape as *Rotating the Box*. Practise it once.

Multi-source BFS — Rotting Oranges (LC 994)

The other archetype: spread on a grid, count steps. Note the `for _ in range(len(q))` level-by-level trick — that's how you count minutes.

```
from collections import deque

def orangesRotting(grid):
    R, C = len(grid), len(grid[0])
    q, fresh = deque(), 0
    for r in range(R):
        for c in range(C):
            if grid[r][c] == 2:
                q.append((r, c))
            elif grid[r][c] == 1:
                fresh += 1

    minutes = 0
    while q and fresh:
        for _ in range(len(q)):
            # one whole level = one minute
            r, c = q.popleft()
            for dr, dc in ((1,0),(-1,0),(0,1),(0,-1)):
                nr, nc = r + dr, c + dc
                if 0 <= nr < R and 0 <= nc < C and grid[nr][nc] == 1:
                    grid[nr][nc] = 2
                    # mark on PUSH, not on pop
                    fresh -= 1
                    q.append((nr, nc))
            minutes += 1
    return -1 if fresh else minutes
```

Mark visited when you push, never when you pop — otherwise the same cell enters the queue many times. This is the most common BFS bug.

Grid one-liners worth memorising

```
R, C = len(g), len(g[0])
grid = [[0] * C for _ in range(R)]      # NEVER [[0]*C]*R (aliases rows!)
DIRS = ((1, 0), (-1, 0), (0, 1), (0, -1))
DIRS8 = [(a, b) for a in (-1,0,1) for b in (-1,0,1) if (a, b) != (0, 0)]

def inb(r, c):
    return 0 <= r < R and 0 <= c < C

list(zip(*matrix))                       # transpose
[list(r) for r in zip(*matrix[::-1])]    # rotate 90 clockwise
[list(r) for r in zip(*matrix)[::-1]]    # rotate 90 counter-clockwise
```

4. Sliding window + hash map

Confidence: medium-high. Roblox guides say sliding window shows up “especially in contexts like rate limiting or substring optimization”. The tagged data carries *Longest Substring Without Repeating Characters*, *Count Number of Nice Subarrays*, *Maximum Number of Occurrences of a Substring*, *Subarrays with K Different Integers*. This also blends naturally into bet #1.

Variable window — longest substring without repeats

```
def lengthOfLongestSubstring(s: str) -> int:
    last, left, best = {}, 0, 0
    for right, ch in enumerate(s):
        if ch in last and last[ch] >= left:
            left = last[ch] + 1          # jump, don't crawl
        last[ch] = right
        best = max(best, right - left + 1)
    return best
```

The template that solves half of them: “at most K”

Memorise this shape. `total += right - left + 1` counts every valid window ending at right.

```
from collections import defaultdict

def at_most_k_distinct(s, k):
    cnt, left, total = defaultdict(int), 0, 0
    for right, ch in enumerate(s):
        cnt[ch] += 1
        while len(cnt) > k:
            cnt[s[left]] -= 1
            if cnt[s[left]] == 0:
                del cnt[s[left]]          # must delete, else len() lies
            left += 1
        total += right - left + 1
    return total

# exactly K == at_most(K) - at_most(K - 1)    <- the standard trick
```

Prefix-count — when the window won't slide

If the constraint isn't monotonic, count prefixes in a hash map instead. *Count Number of Nice Subarrays*, *Subarray Sum Equals K*, *Count Nice Pairs* are all this.

```
from collections import defaultdict

def numberOfSubarrays(nums, k):
    # subarrays with exactly k odds
    seen = defaultdict(int)
    seen[0] = 1                    # empty prefix: essential
    odds = res = 0
    for n in nums:
        odds += n & 1
        res += seen[odds - k]
```

```
        seen[odds] += 1
    return res

def subarraySum(nums, k):                # exactly the same skeleton
    seen = defaultdict(int); seen[0] = 1
    total = res = 0
    for n in nums:
        total += n
        res += seen[total - k]
        seen[total] += 1
    return res
```

Forgetting `seen[0] = 1` is the classic bug — it's what lets a prefix that itself satisfies the condition count.

5. Heap, greedy scheduling, intervals

Confidence: medium. *Reorganize String* (94.7) and *Task Scheduler* (85.6) rank near the top all-time, and they're the *same problem*: place items with a cooldown, always taking the most frequent first. Both are throttling problems, which is thematically on point. *Merge Intervals*, *Employee Free Time* and *Number of Flowers in Full Bloom* also appear.

Task Scheduler (LC 621)

Lead with the $O(n)$ formula, then offer the heap version if they want the actual ordering.

```
from collections import Counter

def leastInterval(tasks, n):
    c = Counter(tasks)
    mx = max(c.values())
    n_max = sum(1 for v in c.values() if v == mx)
    # (mx-1) full frames of width (n+1), plus the final row of maxima
    return max(len(tasks), (mx - 1) * (n + 1) + n_max)
```

Reorganize String (LC 767) — heap with a held-back slot

```
import heapq
from collections import Counter

def reorganizeString(s: str) -> str:
    heap = [(-v, ch) for ch, v in Counter(s).items()]
    heapq.heapify(heap)                                # max-heap via negated counts
    res, prev = [], None
    while heap:
        cnt, ch = heapq.heappop(heap)
        res.append(ch)
        if prev:                                     # re-arm the previous char now
            heapq.heappush(heap, prev)
        cnt += 1                                     # cnt is negative; += shrinks it
        prev = (cnt, ch) if cnt else None
    out = "".join(res)
    return out if len(out) == len(s) else ""
```

The pattern: **pop the best, hold it aside for one turn, push it back**. That single idea covers cooldown scheduling generally.

Intervals

```
def merge(intervals):
    intervals.sort()                                # sorts by start, then end
    out = []
    for s, e in intervals:
        if out and s <= out[-1][1]:                # use < if touching != overlapping
            out[-1][1] = max(out[-1][1], e)
        else:
            out.append([s, e])
    return out
```

```

def max_concurrent(intervals):                # sweep line / "meeting rooms II"
    events = []
    for s, e in intervals:
        events.append((s, 1))
        events.append((e, -1))
    events.sort()                             # at equal time -1 sorts first, so
    cur = best = 0                             # touching intervals don't stack
    for _, delta in events:
        cur += delta
        best = max(best, cur)
    return best

```

Always ask: **does [1,3] overlap [3,5]**? Half-open vs closed intervals is the #1 source of wrong answers here, and asking scores clarification points.

6. Graph / topological sort

Confidence: medium. *Course Schedule II* is #7 all-time; Roblox guides call graphs the heaviest DSA emphasis. For Safety Foundation this maps directly onto **moderation workflow dependencies** — “these checks must run before those; detect a cycle in the pipeline config”. If the problem is dressed as a workflow/DAG, this is it.

```
from collections import deque, defaultdict

def findOrder(n, prerequisites):
    graph = defaultdict(list)
    indeg = [0] * n
    for course, pre in prerequisites:    # CHECK THE ORDER OF THIS PAIR
        graph[pre].append(course)
        indeg[course] += 1

    q = deque(i for i in range(n) if indeg[i] == 0)
    order = []
    while q:
        node = q.popleft()
        order.append(node)
        for nxt in graph[node]:
            indeg[nxt] -= 1
            if indeg[nxt] == 0:
                q.append(nxt)
    return order if len(order) == n else []    # [] means a cycle exists
```

Kahn's algorithm. `len(order) != n` is your cycle detector — say that out loud, it's the point of the problem. $O(V + E)$.

Plain flood fill — Number of Islands (LC 200)

```
def numIslands(grid):
    R, C, count = len(grid), len(grid[0]), 0

    def sink(r, c):    # iterative: no stack-depth risk
        stack = [(r, c)]
        grid[r][c] = "0"
        while stack:
            i, j = stack.pop()
            for di, dj in ((1,0), (-1,0), (0,1), (0,-1)):
                ni, nj = i + di, j + dj
                if 0 <= ni < R and 0 <= nj < C and grid[ni][nj] == "1":
                    grid[ni][nj] = "0"
                    stack.append((ni, nj))

    for r in range(R):
        for c in range(C):
            if grid[r][c] == "1":
                count += 1
                sink(r, c)
    return count
```

Recursion limit is 1000, so a recursive DFS can blow the stack on a big grid. Go iterative (above), or call `sys.setrecursionlimit(10**6)`.

7. Hash-map aggregation (likely warm-up)

Confidence: medium, but nearly free. *Subdomain Visit Count* ranks #1 by raw frequency in one Roblox dataset; *Two Sum* sits at 74.3 in the last six months. If they open with a 5-minute warm-up, it's one of these. Don't overthink them — just be fast and clean.

```
from collections import defaultdict
```

```
def subdomainVisits(cpdomains):          # "900 google.mail.com"
    count = defaultdict(int)
    for cp in cpdomains:
        num, domain = cp.split()
        num = int(num)
        parts = domain.split(".")
        for i in range(len(parts)):      # every suffix is a subdomain
            count[".".join(parts[i:])] += num
    return [f"{v} {k}" for k, v in count.items()]

def twoSum(nums, target):
    seen = {}                           # value -> index
    for i, n in enumerate(nums):
        if target - n in seen:
            return [seen[target - n], i]
        seen[n] = i

def longestConsecutive(nums):            # O(n), not O(n log n)
    s, best = set(nums), 0
    for n in s:
        if n - 1 not in s:               # only start counting at run heads
            m = n
            while m + 1 in s:
                m += 1
            best = max(best, m - n + 1)
    return best

def groupThePeople(sizes):
    buckets, res = defaultdict(list), []
    for person, size in enumerate(sizes):
        buckets[size].append(person)
        if len(buckets[size]) == size:   # full -> ship it, start fresh
            res.append(buckets[size])
            buckets[size] = []
    return res
```

Python cheat sheet

You said you're rusty. This is the 90% of standard library you'll actually reach for. Read this bit twice.

The import block, from memory

```
from collections import defaultdict, Counter, deque, OrderedDict
import heapq
import bisect
from functools import lru_cache, cache, cmp_to_key
from itertools import accumulate, combinations, permutations, groupby
import math, sys
```

collections — you will use these three

Thing	Why / gotcha
<code>defaultdict(list)</code> <code>defaultdict(int)</code> <code>defaultdict(set)</code>	Auto-creates the missing value. <code>d[k].append(x)</code> just works. Gotcha: merely <i>reading</i> <code>d[k]</code> creates the key — use <code>d.get(k)</code> if you don't want that.
<code>Counter(xs)</code>	<code>.most_common(k)</code> returns <code>[(item, n), ...]</code> sorted desc. Supports <code>+</code> <code>-</code> <code>&</code> <code> </code> between Counters. <code>c.total()</code> (3.10+). Missing key returns 0, never <code>KeyError</code> .
<code>deque()</code>	O(1) at both ends: <code>append</code> / <code>appendleft</code> / <code>pop</code> / <code>popleft</code> . <code>deque(maxlen=n)</code> auto-evicts from the left — a free sliding window. Never use <code>list.pop(0)</code> , that's <code>O(n)</code> .
<code>OrderedDict()</code>	<code>.move_to_end(k)</code> and <code>.popitem(last=False)</code> give you an LRU cache in ~10 lines. Plain dicts keep insertion order but lack these.

heapq — min-heap only

```
h = []
heapq.heappush(h, x)
smallest = heapq.heappop(h)
heapq.heapify(existing_list)           # O(n), in place
heapq.heappushpop(h, x)                # push then pop, faster
heapq.nlargest(k, xs, key=lambda t: t[1])
heapq.nsmallest(k, xs)
h[0]                                   # peek, does not remove

# MAX-HEAP: push negatives.
heapq.heappush(h, -count)              # then negate again on pop

# Tuples compare left-to-right. If ties can reach an unorderable
# object, add a monotonic counter as a tiebreak:
heapq.heappush(h, (priority, next(counter), payload_object))
```

bisect — binary search you don't have to write

```
bisect.bisect_left(a, x)               # first index i where a[i] >= x
```

```

bisect.bisect_right(a, x)    # first index i where a[i] > x
bisect.insort(a, x)         # insert keeping order (O(n) memmove)

# count of x in sorted a:
bisect.bisect_right(a, x) - bisect.bisect_left(a, x)

# Python 3.10+: search by key without decorating
bisect.bisect_left(rows, t, key=lambda r: r[0])

```

If you catch yourself hand-writing while `lo < hi`, reach for `bisect` instead — fewer off-by-ones under pressure. If you must hand-roll: `mid = (lo + hi) // 2`, and for “minimum feasible value” problems use `if ok(mid): hi = mid` else: `lo = mid + 1`, answer is `lo`.

Sorting

```

xs.sort()                                # in place
ys = sorted(xs, reverse=True)
xs.sort(key=lambda t: (t[0], -t[1]))     # asc by [0], desc by [1]
xs.sort(key=lambda s: (-len(s), s))     # long first, then A-Z

# descending by a non-numeric key -> use cmp_to_key or sort twice
# (Python sort is stable, so two passes works and is easy to explain)

```

Strings

```

s.split()          # on any whitespace, drops empties
s.split(".")       # on a literal
"".join(parts)
s.lower() / .upper() / .strip() / .replace(a, b)
s.startswith(p) / s.endswith(x)
s.isalnum() / .isalpha() / .isdigit() / .isspace()
ord(c) - ord("a")  # char -> 0..25          chr(97) -> "a"
s[::-1]           # reverse
f"{count} {name}"  # f-string

# Strings are immutable: build with a list, join at the end.
out = []; out.append(ch); "".join(out)

```

Idioms that save you minutes

```

for i, x in enumerate(xs, start=1): ...
for a, b in zip(xs, ys): ...
for a, b in zip(xs, xs[1:]): ...      # adjacent pairs
INF = float("inf")                   # or math.inf
q, r = divmod(a, b)
any(...) / all(...) / sum(1 for x in xs if pred(x))
prefix = list(accumulate(nums, initial=0)) # prefix[i+1]-prefix[j] sums
xs[i], xs[j] = xs[j], xs[i]          # swap
seen = set(); seen.add(x); x in seen # O(1) membership

@cache                               # 3.9+; else @lru_cache(None)
def solve(i, j): ...                 # instant memoised recursion

while q and q[0] <= t - W: q.popleft() # the window-expiry line
sys.setrecursionlimit(10 ** 6)        # before deep DFS

```

The aliasing bug that ruins grid problems: `[[0] * C] * R` makes R references to the *same* row. Always write `[[0] * C for _ in range(R)]`.

Complexity quick reference

Operation	Cost
dict / set get, set, in	$O(1)$ average
list.append / pop() from the end	$O(1)$ amortised
list.pop(0) / insert(0, x) / in on a list	$O(n)$ — use a deque or a set
deque append/pop, either end	$O(1)$
heappush / heappop	$O(\log n)$
heapify	$O(n)$
sorted() / .sort()	$O(n \log n)$
bisect_*	$O(\log n)$
Trie insert / lookup	$O(\text{len}(\text{word}))$
BFS / DFS over a graph	$O(V + E)$
BFS / DFS over an $R \times C$ grid	$O(R \cdot C)$

How to actually run the 45 minutes

Clarifying questions that always earn points

- **Scale:** “How large can the input get? Does it fit in memory?” This is the question that unlocks the bucketing/eviction follow-up on its own.
- **Ordering:** “Are timestamps non-decreasing? Can two events share a timestamp?” (For every stream problem: yes, they can. Handle it.)
- **Duplicates & empties:** empty input, single element, all identical.
- **Concurrency:** “Is this called from one thread or many?” — for a platform team this is a genuinely good question, not a stall.
- **Semantics:** for intervals, does touching count as overlapping? For text, case-sensitive? Unicode?

What a senior engineer is scoring

At this level correctness is table stakes. The differentiators, roughly in the order they're weighted:

1. **You stated the approach and its complexity before typing.** Nothing else moves the needle as much.
2. **You picked a data structure and said why.** “Deque, because I need $O(1)$ eviction from the front as the window slides.”
3. **Your code is readable.** Real names, small functions, no single-letter variables outside loop indices. Roblox guides call out clean, testable code specifically.
4. **You tested it yourself** with a concrete tiny input, walking the state, before they asked. Find your own bug — that's a huge positive.
5. **You raised the production concern unprompted:** memory growth, thread safety, what breaks across many servers, what happens on clock skew.
6. **You were pleasant to think with.** 45 minutes is also a “would I want this person on my team” test.

If you get stuck: say what you know is true, propose the brute force, state its complexity, then optimise out loud. A working brute force with a clear path to better beats silence every single time. Never go quiet for more than ~20 seconds — narrate even the dead ends.

Things not to do

- Don't start coding in minute two. It reads as junior even if you're right.
- Don't silently rewrite. If you change approach, say why.

- Don't claim a complexity you haven't checked — they will probe it.
- Don't skip the dry run to save time. Budget the last 8 minutes for it.
- Don't ignore the product framing. If they wrapped it in a chat/moderation story, use their vocabulary in your variable names. It shows you were listening.

Two sentences to have ready for “any questions for me?”

Ask something only this team could answer. For example: *“The platform is moving to AI-first, agentic moderation workflows — how do you keep those auditable and debuggable when a decision is made by a model rather than a rule?”* Or: *“You're targeting 10x traffic while cutting integration time from months to hours — which of those two is harder right now?”*

Both come straight out of the team's own description of its work, and both are real engineering questions rather than culture filler.

Roblox values, for the last five minutes

Roblox weaves values into every round. If a behavioural question sneaks in, these are the hooks: **Respect the Community** · **We are Responsible** (owning unintended consequences — very Safety) · **Take the Long View** · **Get Stuff Done**. Have one story where you chose the durable fix over the quick one, and one where a decision of yours had a consequence you didn't foresee and you owned it.

Good luck tomorrow. Type out the rate limiter and the trie once each before bed — muscle memory on those two covers most of the probability mass.