WISE ROUND 2 MANAGER PREP — REMAINING 3 PROBLEMS

Cache with TTL → LRU, HTTP Client with Retry, Merge Intervals

Same format: Descriptive + Code + What to Say + Tests

PROBLEM 3: CACHE WITH TTL → EXTEND TO LRU

Confirmed: Glassdoor Mar 2025 (Tallinn) — "a simple cache programming"

How the Interview Flows

They'll give it to you in stages. Don't jump ahead — solve each stage, then they'll add the next requirement.

Stage 1: "Build a simple key-value cache" **Stage 2:** "Add a maximum size — what happens when it's full?" **Stage 3:** "Add TTL — entries should expire after N seconds" **Stage 4:** "How would you make this thread-safe?"

Stage 1: Simple Key-Value Cache

The interviewer says: "*Build a simple cache that stores and retrieves values by key.*"

Start by clarifying: "Should the cache be generic — work with any key and value types? And should I think about thread safety now or later?"

They'll likely say: "Start simple, we'll add requirements."

swift

```swift
// "I'll make this generic from the start because it's no extra effort
//  and saves us from rewriting when we need different types."

class Cache<Key: Hashable, Value> {
    private var storage: [Key: Value] = [:]

    func get(_ key: Key) -> Value? {
        storage[key]
    }

    func set(_ key: Key, value: Value) {
        storage[key] = value
    }

    func remove(_ key: Key) {
        storage.removeValue(forKey: key)
    }

    var count: Int { storage.count }
}
```

What to SAY:

> "This is the simplest implementation — a dictionary wrapper. It's O(1) for get, set, and remove. In production I wouldn't ship just this, but it's the right starting point. What requirements should I add?"

Stage 2: Add Maximum Size with LRU Eviction

The interviewer says: "*The cache should have a maximum size. When it's full and we add a new item, remove the least recently used entry.*"

> "So we need an eviction policy. LRU means we track access order — every time we get or set a key, it becomes the most recently used. When we're full, we evict the entry that hasn't been accessed for the longest time.
> The classic implementation uses a doubly linked list for O(1) move-to-front and removal, combined with a dictionary for O(1) key lookup. The list maintains access order, the dictionary provides fast key access. Together, all operations are O(1)."

```swift
```

```
class LRUCache<Key: Hashable, Value> {

    // Doubly linked list node
    private class Node {
        let key: Key
        var value: Value
        var prev: Node?
        var next: Node?

        init(key: Key, value: Value) {
            self.key = key
            self.value = value
        }
    }

    private let capacity: Int
    private var cache: [Key: Node] = [:]

    // Dummy head and tail simplify edge cases — no nil checks needed
    private let head = Node(key: "" as! Key, value: "" as! Value) // sentinel
    private let tail = Node(key: "" as! Key, value: "" as! Value) // sentinel

    init(capacity: Int) {
        self.capacity = capacity
        head.next = tail
        tail.prev = head
    }

    func get(_ key: Key) -> Value? {
        guard let node = cache[key] else { return nil }
        moveToFront(node)  // Mark as recently used
        return node.value
    }

    func set(_ key: Key, value: Value) {
        if let node = cache[key] {
            // Key exists — update value and move to front
            node.value = value
            moveToFront(node)
        } else {
            // New key — create node, add to front
            let newNode = Node(key: key, value: value)
            cache[key] = newNode
            addAfterHead(newNode)

            // Evict if over capacity
```

```
            if cache.count > capacity {
                let lru = tail.prev!  // Least recently used = just before tail
                removeNode(lru)
                cache.removeValue(forKey: lru.key)
            }
        }
    }

    // ── Linked list operations (all O(1)) ──

    private func addAfterHead(_ node: Node) {
        node.prev = head
        node.next = head.next
        head.next?.prev = node
        head.next = node
    }

    private func removeNode(_ node: Node) {
        node.prev?.next = node.next
        node.next?.prev = node.prev
    }

    private func moveToFront(_ node: Node) {
        removeNode(node)
        addAfterHead(node)
    }
}
```

What to SAY while coding:

> *Writing the Node class:* "I'm using a doubly linked list because we need O(1) removal from any position. A singly linked list would require O(n) to find the previous node."

> *Writing dummy head/tail:* "Sentinel nodes eliminate edge cases. Without them, every operation needs null checks for 'is this the first node?' or 'is this the last node?' With sentinels, the list is never truly empty, and every real node always has valid prev and next pointers."

> *On eviction:* "The node just before the tail is always the LRU — the one that hasn't been touched the longest. Evicting it is O(1): remove from list, remove from dictionary."

> *On* `get` *moving to front:* "This is key — every access updates recency. If we only tracked insertion order, we'd have FIFO, not LRU."

Stage 3: Add TTL (Time-To-Live)

The interviewer says: "*Entries should expire after a configurable number of seconds*."

> "I'll add a timestamp and TTL to each node. When we `get`, we check if the entry has expired. If it has, we treat it as a cache miss and remove it. I'll also inject a Clock protocol so this is testable without waiting real seconds."

swift

```
protocol Clock {
    func now() -> Date
}

struct SystemClock: Clock {
    func now() -> Date { Date() }
}

class LRUCacheWithTTL<Key: Hashable, Value> {

    private class Node {
        let key: Key
        var value: Value
        let createdAt: Date
        var prev: Node?
        var next: Node?

        init(key: Key, value: Value, createdAt: Date) {
            self.key = key
            self.value = value
            self.createdAt = createdAt
        }

        func isExpired(now: Date, ttl: TimeInterval) -> Bool {
            now.timeIntervalSince(createdAt) > ttl
        }
    }

    private let capacity: Int
    private let ttl: TimeInterval
    private let clock: Clock
    private var cache: [Key: Node] = [:]
    private let head: Node
    private let tail: Node

    init(capacity: Int, ttlSeconds: TimeInterval, clock: Clock = SystemClock()) {
        self.capacity = capacity
        self.ttl = ttlSeconds
        self.clock = clock
        // Initialize sentinels (using placeholder values)
        self.head = Node(key: unsafeBitCast(0, to: Key.self),
                         value: unsafeBitCast(0, to: Value.self),
                         createdAt: Date.distantPast)
        self.tail = Node(key: unsafeBitCast(0, to: Key.self),
                         value: unsafeBitCast(0, to: Value.self),
                         createdAt: Date.distantPast)
```

```
    head.next = tail
    tail.prev = head
}

func get(_ key: Key) -> Value? {
    guard let node = cache[key] else { return nil }

    // Check TTL
    if node.isExpired(now: clock.now(), ttl: ttl) {
        // Expired — remove and return nil (cache miss)
        removeNode(node)
        cache.removeValue(forKey: key)
        return nil
    }

    moveToFront(node)
    return node.value
}

func set(_ key: Key, value: Value) {
    if let existing = cache[key] {
        removeNode(existing)
        cache.removeValue(forKey: key)
    }

    let node = Node(key: key, value: value, createdAt: clock.now())
    cache[key] = node
    addAfterHead(node)

    // Evict if over capacity
    while cache.count > capacity {
        let lru = tail.prev!
        removeNode(lru)
        cache.removeValue(forKey: lru.key)
    }
}

// Same linked list helpers as before
private func addAfterHead(_ node: Node) {
    node.prev = head; node.next = head.next
    head.next?.prev = node; head.next = node
}
private func removeNode(_ node: Node) {
    node.prev?.next = node.next; node.next?.prev = node.prev
}
private func moveToFront(_ node: Node) {
    removeNode(node); addAfterHead(node)
```

```
    }
}
```

What to SAY:

> "I chose lazy expiration — we check TTL on access, not proactively. This is simpler and avoids background timers. The trade-off is that expired entries occupy memory until accessed. If memory is critical, we could add a periodic cleanup, but for a rate cache with small entries, lazy expiration is the pragmatic choice."

> "I inject the Clock so tests can control time. Without this, testing a 30-second TTL requires sleeping 30 real seconds. With a MockClock, I advance time programmatically — test runs in milliseconds."

Stage 4: Thread Safety Discussion

They ask: "*How would you make this thread-safe?*"

> "Three options:
> **Option A — Actor (modern Swift, recommended):** Wrap the cache as an actor. All access automatically serialized. Callers use `await`. This is the most Swifty approach and the compiler enforces correctness — you can't accidentally access the cache without `await`.

```swift
actor ThreadSafeCache<Key: Hashable, Value> {
    private let cache: LRUCacheWithTTL<Key, Value>

    init(capacity: Int, ttlSeconds: TimeInterval) {
        cache = LRUCacheWithTTL(capacity: capacity, ttlSeconds: ttlSeconds)
    }

    func get(_ key: Key) -> Value? { cache.get(key) }
    func set(_ key: Key, value: Value) { cache.set(key, value: value) }
}
// Usage: await cache.get("USD-EUR")
```

> **Option B — Serial DispatchQueue (older codebases):** Wrap all reads and writes in a serial queue. Simpler to retrofit into existing code.

```swift
```

```
class ThreadSafeCacheGCD<Key: Hashable, Value> {
    private let cache: LRUCacheWithTTL<Key, Value>
    private let queue = DispatchQueue(label: "com.wise.ratecache")

    func get(_ key: Key) -> Value? {
        queue.sync { cache.get(key) }
    }

    func set(_ key: Key, value: Value) {
        queue.sync { cache.set(key, value: value) }
    }
}
```

> **Option C — Read-Write Lock (performance-optimized):** Allow concurrent reads but exclusive writes. Best for read-heavy caches.

> "For Wise's rate cache, I'd use the actor approach — it's modern, compiler-enforced, and Wise already uses Swift concurrency. The read-write lock is more performant for extreme read:write ratios, but the actor is simpler and correctness matters more than micro-optimization in fintech."

Tests (Manager Signal)

swift

```
class MockClock: Clock {
    var current = Date()
    func now() -> Date { current }
    func advance(by seconds: TimeInterval) {
        current = current.addingTimeInterval(seconds)
    }
}

func testCache_basicGetSet() {
    let cache = LRUCacheWithTTL<String, Decimal>(capacity: 3, ttlSeconds: 30)
    cache.set("USD-EUR", value: Decimal(string: "0.92")!)
    XCTAssertEqual(cache.get("USD-EUR"), Decimal(string: "0.92")!)
}

func testCache_evictsLRU() {
    let cache = LRUCacheWithTTL<String, Int>(capacity: 2, ttlSeconds: 60)
    cache.set("a", value: 1)
    cache.set("b", value: 2)
    cache.set("c", value: 3)  // "a" should be evicted

    XCTAssertNil(cache.get("a"))       // Evicted
    XCTAssertEqual(cache.get("b"), 2)  // Still there
    XCTAssertEqual(cache.get("c"), 3)  // Still there
}

func testCache_expiresAfterTTL() {
    let mockClock = MockClock()
    let cache = LRUCacheWithTTL<String, Decimal>(
        capacity: 10, ttlSeconds: 30, clock: mockClock
    )
    cache.set("USD-EUR", value: Decimal(string: "0.92")!)

    XCTAssertNotNil(cache.get("USD-EUR"))  // Fresh — exists

    mockClock.advance(by: 31)  // 31 seconds later

    XCTAssertNil(cache.get("USD-EUR"))  // Expired — gone
}

func testCache_accessUpdatesRecency() {
    let cache = LRUCacheWithTTL<String, Int>(capacity: 2, ttlSeconds: 60)
    cache.set("a", value: 1)
    cache.set("b", value: 2)

    _ = cache.get("a")  // "a" is now most recent
```

```
    cache.set("c", value: 3)  // "b" should be evicted (it's LRU), not "a"

    XCTAssertEqual(cache.get("a"), 1)  // Still here — was accessed recently
    XCTAssertNil(cache.get("b"))       // Evicted — wasn't accessed
}
```

PROBLEM 4: HTTP CLIENT → ADD RETRY → ADD CIRCUIT BREAKER

Confirmed: Reddit — "I bombed... wasn't expecting HTTP requests"

How the Interview Flows

Stage 1: "Build a simple HTTP client that fetches data from a URL" **Stage 2:** "Add retry logic with exponential backoff" **Stage 3:** "Some errors shouldn't be retried. Handle that." **Stage 4:** "Add concurrent request support"

Stage 1: Simple HTTP Client

> "I'll build this with async/await and a protocol for the underlying session so it's testable."

swift

```swift
// Protocol for testability
protocol NetworkSession {
    func data(from url: URL) async throws -> (Data, URLResponse)
}
extension URLSession: NetworkSession {}

// Typed errors — the caller needs to know WHAT went wrong
enum HTTPError: Error {
    case invalidURL(String)
    case badStatusCode(Int)
    case noData
    case decodingFailed(Error)
    case networkFailure(Error)
}

class HTTPClient {
    private let session: NetworkSession

    init(session: NetworkSession = URLSession.shared) {
        self.session = session
    }

    func fetch<T: Decodable>(_ type: T.Type, from urlString: String) async throws -> T {
        guard let url = URL(string: urlString) else {
            throw HTTPError.invalidURL(urlString)
        }

        let (data, response) = try await session.data(from: url)

        guard let httpResponse = response as? HTTPURLResponse else {
            throw HTTPError.noData
        }

        guard (200...299).contains(httpResponse.statusCode) else {
            throw HTTPError.badStatusCode(httpResponse.statusCode)
        }

        do {
            return try JSONDecoder().decode(T.self, from: data)
        } catch {
            throw HTTPError.decodingFailed(error)
        }
    }
}
```

What to SAY:

> "I'm making `fetch` generic over `T: Decodable` so the same client works for any response type — rates, transfers, balances. The caller says `fetch(Rate.self, from: url)` and gets a typed result.
> I separate status code validation from decoding because a 400 error with a valid JSON body shouldn't throw a decoding error — it should throw a `badStatusCode` so the caller knows it was a client error."

Stage 2: Add Retry with Exponential Backoff

The interviewer says: "*Network requests sometimes fail transiently. Add retry logic.*"

> "Exponential backoff means we wait longer between each retry — 1 second, 2 seconds, 4 seconds. This prevents overwhelming a struggling server with rapid retries. I'll add optional jitter to avoid the thundering herd problem where all clients retry at the same moment."

```swift
```

```
struct RetryConfig {
    let maxAttempts: Int
    let initialDelay: TimeInterval
    let multiplier: Double
    let maxDelay: TimeInterval

    static let `default` = RetryConfig(
        maxAttempts: 3,
        initialDelay: 1.0,
        multiplier: 2.0,
        maxDelay: 10.0
    )
}

class RetryHTTPClient {
    private let session: NetworkSession
    private let retryConfig: RetryConfig

    init(session: NetworkSession = URLSession.shared,
         retryConfig: RetryConfig = .default) {
        self.session = session
        self.retryConfig = retryConfig
    }

    func fetch<T: Decodable>(_ type: T.Type, from urlString: String) async throws -> T {
        guard let url = URL(string: urlString) else {
            throw HTTPError.invalidURL(urlString)
        }

        var lastError: Error?

        for attempt in 0..<retryConfig.maxAttempts {
            do {
                let (data, response) = try await session.data(from: url)

                guard let httpResponse = response as? HTTPURLResponse else {
                    throw HTTPError.noData
                }

                // Don't retry client errors (4xx) — they won't fix themselves
                if (400...499).contains(httpResponse.statusCode) {
                    throw HTTPError.badStatusCode(httpResponse.statusCode)
                }

                // Retry server errors (5xx)
                if httpResponse.statusCode >= 500 {
```

```
            throw HTTPError.badStatusCode(httpResponse.statusCode)
        }

        guard (200...299).contains(httpResponse.statusCode) else {
            throw HTTPError.badStatusCode(httpResponse.statusCode)
        }

        return try JSONDecoder().decode(T.self, from: data)

    } catch let error as HTTPError {
        lastError = error

        // Don't retry non-retryable errors
        guard isRetryable(error) else { throw error }

        // Don't sleep after the last attempt
        if attempt < retryConfig.maxAttempts - 1 {
            let delay = calculateDelay(attempt: attempt)
            try await Task.sleep(nanoseconds: UInt64(delay * 1_000_000_000))
        }
    } catch {
        lastError = error

        if attempt < retryConfig.maxAttempts - 1 {
            let delay = calculateDelay(attempt: attempt)
            try await Task.sleep(nanoseconds: UInt64(delay * 1_000_000_000))
        }
    }
}

throw lastError ?? HTTPError.noData
}

private func isRetryable(_ error: HTTPError) -> Bool {
    switch error {
    case .badStatusCode(let code):
        return code >= 500   // Retry server errors
    case .networkFailure:
        return true          // Retry network issues
    case .invalidURL, .decodingFailed, .noData:
        return false         // Don't retry — won't fix themselves
    }
}

private func calculateDelay(attempt: Int) -> TimeInterval {
    let delay = retryConfig.initialDelay * pow(retryConfig.multiplier, Double(attempt))
    let capped = min(delay, retryConfig.maxDelay)
```

```
        // Add jitter (±25%) to prevent thundering herd
        let jitter = capped * Double.random(in: 0.75...1.25)
        return jitter
    }
}
```

What to SAY:

> *On 4xx vs 5xx:* "Critical distinction — 4xx errors are client errors. If I send an invalid amount, retrying the same request will get the same 400 error every time. 5xx errors are server errors — the server might recover, so retry makes sense. At PayPal, I've seen bugs where we retried 401 Unauthorized infinitely because someone forgot this distinction."

> *On exponential backoff:* "First retry after 1 second, second after 2 seconds, third after 4 seconds. Each wait doubles. We cap at 10 seconds because making a user wait longer than that is worse than showing an error."

> *On jitter:* "Without jitter, if 10,000 users all get a 500 error at the same time, they all retry exactly 1 second later, causing another spike. Jitter randomises the retry window so retries are spread out. This is standard practice at scale."

> *On Task.sleep:* "I use `Task.sleep` which respects task cancellation — if the user navigates away, the task is cancelled and we don't continue retrying for a screen that's no longer visible."

Stage 3: Add Concurrent Requests

The interviewer says: "*What if you need to fetch from multiple URLs simultaneously?*"

swift

```swift
extension RetryHTTPClient {

    // Fetch multiple URLs concurrently, return all results
    func fetchAll<T: Decodable>(
        _ type: T.Type,
        from urls: [String]
    ) async throws -> [T] {
        try await withThrowingTaskGroup(of: T.self) { group in
            for url in urls {
                group.addTask {
                    try await self.fetch(type, from: url)
                }
            }

            var results: [T] = []
            for try await result in group {
                results.append(result)
            }
            return results
        }
    }

    // Fetch with timeout
    func fetchWithTimeout<T: Decodable>(
        _ type: T.Type,
        from urlString: String,
        timeoutSeconds: TimeInterval
    ) async throws -> T {
        try await withThrowingTaskGroup(of: T.self) { group in
            group.addTask {
                try await self.fetch(type, from: urlString)
            }
            group.addTask {
                try await Task.sleep(nanoseconds: UInt64(timeoutSeconds * 1_000_000_000))
                throw HTTPError.networkFailure(
                    URLError(.timedOut)
                )
            }

            // First to complete wins — either result or timeout
            let result = try await group.next()!
            group.cancelAll()  // Cancel the other task
            return result
        }
```

```
    }
}
```

What to SAY:

> "TaskGroup gives us structured concurrency — if any task fails, we can handle it. If the parent task is cancelled, all children are automatically cancelled. No orphaned network requests.
> For the timeout, I race the actual fetch against a sleep task. Whichever finishes first wins. If the fetch completes before the timeout, we cancel the sleep. If the sleep completes first, we throw a timeout error and cancel the fetch. This is a clean pattern for timeouts in structured concurrency."

Tests

swift

```
class MockSession: NetworkSession {
    var mockData: Data = Data()
    var mockStatusCode: Int = 200
    var shouldFail: Bool = false
    var callCount = 0

    func data(from url: URL) async throws -> (Data, URLResponse) {
        callCount += 1
        if shouldFail { throw URLError(.notConnectedToInternet) }
        let response = HTTPURLResponse(url: url, statusCode: mockStatusCode,
                                       httpVersion: nil, headerFields: nil)!
        return (mockData, response)
    }
}

func testFetch_success() async throws {
    let mock = MockSession()
    mock.mockData = """{"rate": 0.92}""".data(using: .utf8)!
    let client = RetryHTTPClient(session: mock, retryConfig: .default)

    struct Rate: Decodable { let rate: Decimal }
    let result = try await client.fetch(Rate.self, from: "https://api.wise.com/rate")

    XCTAssertEqual(result.rate, Decimal(string: "0.92"))
    XCTAssertEqual(mock.callCount, 1)  // No retries needed
}

func testFetch_retriesOnServerError() async throws {
    let mock = MockSession()
    mock.mockStatusCode = 500
    let config = RetryConfig(maxAttempts: 3, initialDelay: 0.01,
                             multiplier: 2.0, maxDelay: 0.1)
    let client = RetryHTTPClient(session: mock, retryConfig: config)

    struct Rate: Decodable { let rate: Decimal }
    do {
        _ = try await client.fetch(Rate.self, from: "https://api.wise.com/rate")
        XCTFail("Should have thrown")
    } catch {
        XCTAssertEqual(mock.callCount, 3)  // Retried 3 times
    }
}

func testFetch_doesNotRetryClientError() async throws {
    let mock = MockSession()
    mock.mockStatusCode = 400
```

```
    let client = RetryHTTPClient(session: mock)

    struct Rate: Decodable { let rate: Decimal }
    do {
        _ = try await client.fetch(Rate.self, from: "https://api.wise.com/rate")
        XCTFail("Should have thrown")
    } catch {
        XCTAssertEqual(mock.callCount, 1)  // NO retry — 400 is not retryable
    }
}
```

PROBLEM 5: MERGE INTERVALS + WISE CONTEXT

Confirmed: Wise's own blog — "sort 2 lists of intervals"

How the Interview Flows

Stage 1: "Given a list of intervals, merge overlapping ones" **Stage 2:** "Now given TWO lists of intervals, find the intersection" **Stage 3:** "Now insert a new interval into the merged list" **Stage 4:** Discussion on Wise context — "where would this apply?"

Stage 1: Merge Overlapping Intervals (LC 56)

The interviewer says: "*Given a list of intervals, merge all overlapping intervals*."

> "I'll sort by start time first. Then I iterate through — if the current interval's start overlaps with the previous interval's end, I merge them by extending the end. Otherwise, I add the current interval as a new non-overlapping segment."

swift

```swift
func mergeIntervals(_ intervals: [[Int]]) -> [[Int]] {
    guard intervals.count > 1 else { return intervals }

    // Sort by start time
    let sorted = intervals.sorted { $0[0] < $1[0] }

    var result: [[Int]] = [sorted[0]]

    for i in 1..<sorted.count {
        let current = sorted[i]
        let lastMerged = result[result.count - 1]

        if current[0] <= lastMerged[1] {
            // Overlapping — extend the end
            result[result.count - 1][1] = max(lastMerged[1], current[1])
        } else {
            // Not overlapping — add as new interval
            result.append(current)
        }
    }

    return result
}

// Example:
// Input:  [[1,3], [2,6], [8,10], [15,18]]
// Sorted: [[1,3], [2,6], [8,10], [15,18]]
// Step 1: [1,3] starts result
// Step 2: [2,6] overlaps (2 <= 3), merge → [1,6]
// Step 3: [8,10] no overlap (8 > 6), add → [1,6], [8,10]
// Step 4: [15,18] no overlap, add → [1,6], [8,10], [15,18]
```

What to SAY:

> "Time complexity is O(n log n) for the sort, then O(n) for the merge pass. Total O(n log n). Space is O(n) for the result.
> The key insight is that after sorting, we only need to compare each interval with the last interval in our result — we don't need to check all previous intervals. Sorting transforms a potentially $O(n^2)$ problem into O(n log n)."

Stage 2: Interval List Intersections (LC 986)

The interviewer says: "*Now given TWO sorted lists of intervals, find their intersection.*"

> "Since both lists are already sorted, I'll use a two-pointer approach. For each pair of intervals, the intersection is [max of starts, min of ends] — but only if it's valid (start <= end). Then I advance the pointer for whichever interval ends first."

swift

```swift
func intervalIntersection(_ A: [[Int]], _ B: [[Int]]) -> [[Int]] {
    var i = 0, j = 0
    var result: [[Int]] = []

    while i < A.count && j < B.count {
        // Intersection = [max of starts, min of ends]
        let lo = max(A[i][0], B[j][0])
        let hi = min(A[i][1], B[j][1])

        if lo <= hi {
            result.append([lo, hi])
        }

        // Advance the pointer with the earlier end
        if A[i][1] < B[j][1] {
            i += 1
        } else {
            j += 1
        }
    }

    return result
}

// Example:
// A = [[0,2], [5,10], [13,23]]
// B = [[1,5], [8,12], [15,24]]
//
// i=0,j=0: A[0]=[0,2], B[0]=[1,5] → lo=1, hi=2 → [1,2] ✅
//          A ends first (2 < 5), advance i
// i=1,j=0: A[1]=[5,10], B[0]=[1,5] → lo=5, hi=5 → [5,5] ✅
//          B ends first (5 < 10), advance j
// i=1,j=1: A[1]=[5,10], B[1]=[8,12] → lo=8, hi=10 → [8,10] ✅
//          A ends first (10 < 12), advance i
// i=2,j=1: A[2]=[13,23], B[1]=[8,12] → lo=13, hi=12 → invalid (13 > 12)
//          B ends first, advance j
// i=2,j=2: A[2]=[13,23], B[2]=[15,24] → lo=15, hi=23 → [15,23] ✅
```

What to SAY:

> "This is O(m + n) time and O(m + n) space — we visit each interval once. The two-pointer technique works because both lists are sorted. We don't need nested loops.
> The pointer advancement rule is simple: whichever interval ends first can't overlap with any future intervals from the other list, so we're done with it."

Stage 3: Insert Interval (LC 57)

The interviewer says: *"Now insert a new interval into a non-overlapping sorted list, merging if necessary."*

```swift
func insertInterval(_ intervals: [[Int]], _ newInterval: [Int]) -> [[Int]] {
    var result: [[Int]] = []
    var new = newInterval
    var i = 0

    // Phase 1: Add all intervals that come BEFORE the new interval
    while i < intervals.count && intervals[i][1] < new[0] {
        result.append(intervals[i])
        i += 1
    }

    // Phase 2: Merge all intervals that OVERLAP with the new interval
    while i < intervals.count && intervals[i][0] <= new[1] {
        new[0] = min(new[0], intervals[i][0])
        new[1] = max(new[1], intervals[i][1])
        i += 1
    }
    result.append(new)

    // Phase 3: Add all intervals that come AFTER
    while i < intervals.count {
        result.append(intervals[i])
        i += 1
    }

    return result
}
```

What to SAY:

> "I split this into three clean phases: before, overlap, after. This is O(n) time because we visit each interval exactly once. The merge phase handles the tricky part — expanding the new interval's bounds to encompass all overlapping intervals."

Stage 4: Wise Context Discussion

The interviewer asks: "*Where would interval problems apply at Wise?*"

> "Several places:
> **Transfer processing windows:** Different payment methods have different processing windows. A bank transfer might process between 9 AM - 5 PM in the recipient's timezone. When a user schedules a transfer, we need to determine which processing windows are available — that's an interval intersection problem.
> **Rate validity windows:** Each quoted rate has a validity period. If a user gets multiple quotes at different times, the overlapping validity period is the window where all rates are still valid — that's an intersection.
> **Maintenance windows:** When Wise schedules infrastructure maintenance, we need to merge overlapping maintenance windows to show users a single 'service unavailable' period instead of multiple confusing ones — that's merge intervals.
> **Busy period detection:** If we want to identify peak usage periods from transaction timestamps, we can convert each transaction into a short interval and merge overlapping ones to find busy periods — useful for capacity planning."

EDGE CASES TO MENTION FOR ALL INTERVAL PROBLEMS

swift

```swift
// Edge cases I'd test:
// 1. Empty input
mergeIntervals([])  // → []

// 2. Single interval
mergeIntervals([[1, 3]])  // → [[1, 3]]

// 3. No overlaps
mergeIntervals([[1, 2], [5, 6]])  // → [[1, 2], [5, 6]]

// 4. All overlap into one
mergeIntervals([[1, 4], [2, 5], [3, 6]])  // → [[1, 6]]

// 5. Touching boundaries (1 ends where 2 starts)
mergeIntervals([[1, 3], [3, 5]])  // → [[1, 5]] — touching = overlapping

// 6. One interval completely inside another
mergeIntervals([[1, 10], [3, 5]])  // → [[1, 10]]

// 7. Unsorted input
mergeIntervals([[5, 6], [1, 3], [2, 4]])  // Sort first → [[1, 4], [5, 6]]
```

MANAGER SIGNALS ACROSS ALL 3 PROBLEMS

When you finish ANY problem, volunteer these:

Topic	What to Say
Testing	"I'd test happy path, edge cases (empty, single, all overlap), and the boundaries."
Complexity	State time AND space complexity without being asked.
Production	"In production I'd add logging, metrics, and error monitoring."
Trade-offs	"I chose lazy expiration over proactive cleanup because..."
Team impact	"If a junior wrote this, I'd review for: edge case handling, naming clarity, test coverage."
Wise connection	"This directly applies to rate caching at Wise..." or "transfer processing windows..."