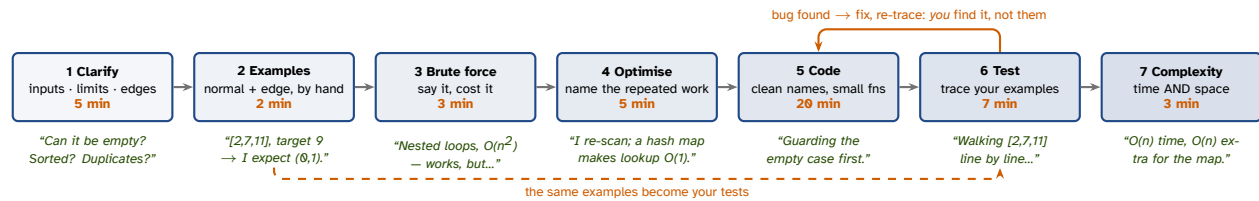


In one line: They grade **how you think, out loud**, not only the final code: **clarify** → **examples** → **brute force (say it)** → **optimise** → **code** → **test with the examples** → **complexity**. A mostly-working solution with clear narration beats a silent perfect one. In iOS exercises the extra marks are for the **platform hygiene**: [weak self], main-thread UI, no !, cancellation, loading/error states.



Communication rules

- **Narrate continuously** — silence is the biggest tell; they cannot help what they cannot hear.
- **State the approach, get a nod, then type**: a wrong turn costs 30 s, not 15 min.
- **Take hints fast**: acknowledge, use it, say why it helps — a hint means they want you to pass.
- **“I don’t know”** → best hypothesis + how you’d verify it. Never bluff.
- **Finished early?** Offer extensions: edge cases, concurrency, tests, an alternative + trade-off.
- Readability is graded: remaining, not r; a helper, not copy-paste.
- **Edge cases to offer unprompted**: empty · one element · duplicates · negatives / zero · huge n (overflow, recursion depth) · already sorted / reversed · emoji in a String · network: slow, offline, 401, malformed JSON, empty page.

iOS pitfalls they watch for

Retain cycles closure stored by something self owns (sink, timer, stored completion) → [weak self] + guard let self; weak var delegate (protocol : AnyObject). Say it aloud.

Main-thread UI dataTask completion runs on a *background* queue → hop to main. Better: @MainActor ViewModel that awaits off-main work.

Force unwraps URL(string:!)!, as!, first! → guard let ... else { throw }. A crash in the demo is fatal.

Codable edge cases snake_case (.convertFromSnakeCase / CodingKeys) · missing key → Optional · **unknown enum case throws** → custom init(from:) with a fallback · number sent as string · .iso8601 rejects fractional seconds.

Cancellation cancel the old search per keystroke and image loads on cell reuse. A cancelled URLSession call throws URLError(.cancelled), *not* CancellationError — swallow both, never show them.

Take-home — what gets graded

- **Builds and runs first time**; no missing keys, few or no deps.
- **README**: how to run · architecture + why · trade-offs · “with more time” · time spent.
- **Structure**: View / ViewModel / Service behind protocols, injected — not a 600-line VC.
- **Tests**: VM with a fake service (loading→loaded, error, empty); decoding from a JSON fixture.
- **States**: loading, empty, error + retry, refresh; Dynamic Type / VoiceOver = bonus.
- Small readable commits; no over-engineering (no DI framework for 3 screens).

Typical iOS exercises → reach for

Exercise	Reach for
Fetch + list	async service → @MainActor VM with a State enum → List / diffable source
Image loading	NSCache → dedupe in-flight by URL → cancel on reuse → check the cell still wants that URL
Debounced search	cancel the previous Task, sleep 300 ms, check isCancelled (or Combine .debounce + removeDuplicates)
Pagination	fetch when a row N from the end appears; guard !isLoading, hasMore; cursor; dedupe by id
Tiny cache	actor + dictionary + LRU order, or NSCache (thread-safe, object keys, evicts under memory pressure)

Skeleton to reach for

```
protocol ItemService { func search(_ q: String) async throws -> [Item] }

@MainActor final class SearchVM: ObservableObject {
    enum State { case idle, loading, loaded([Item]), failed(String) }
    @Published private(set) var state: State = .idle
    private let service: ItemService // injected: fake in tests
    private var task: Task<Void, Never>?
    init(service: ItemService) { self.service = service }

    func queryChanged(_ q: String) {
        task?.cancel() // debounce = cancel + wait
        task = Task { [weak self] in
            try? await Task.sleep(for: .milliseconds(300))
            guard let self, !task?.isCancelled else { return }
            state = .loading
            do { state = .loaded(try await service.search(q)) }
            catch is CancellationError {} // superseded: stay quiet
            catch let e as URLError where e.code == .cancelled {}
            catch { state = .failed(error.localizedDescription) }
        }
    }
}

// paging VM (items, isLoading, hasMore): call from each row's onAppear
func rowAppeared(_ item: Item) {
    guard let i = items.firstIndex(where: { $0.id == item.id }),
        i >= items.count - 5, !isLoading, hasMore else { return }
    Task { await loadNextPage() } // cursor = last page's nextCursor
}

// cell.prepareForReuse(): loadTask?.cancel(); imageView.image = nil
```

Interview traps

- ! Coding in the first 10 s — you solve the wrong problem.
- ! “Done” without tracing; empty input / off-by-one found by *them*.
- ! “O(n)” recited, ignoring the **space** of the map or recursion stack.
- ! Image lands in a *reused* cell → wrong picture: cancel *and* compare the URL.

Remember Clarify, brute force aloud, optimise the repeated work, code, trace your own examples, state time + space.

Likely prompts

1. Build a searchable list from this endpoint. — service + VM + debounce + states.
2. Why [weak self] in that Task? — it would keep the VM alive until the sleep + request finish.
3. How do you test the VM? — inject a fake service, assert the state sequence.
4. More time? — caching, pagination, retry, accessibility.