

Swift Fundamentals & DSA

String

Properties: count, isEmpty, startIndex, endIndex

Case: lowercased(), uppercased()

Search: contains(_:), hasPrefix(_:), hasSuffix(_:), starts(with:)

Transform: reversed(), split(separator:), components(separatedBy:),
trimmingCharacters(in:), replacingOccurrences(of:with:)

Init: String(42), String(3.14), String(repeating:count:)

Character: isLetter, isNumber, isWhitespace, asciiValue

```
s[s.index(s.startIndex, offsetBy: 4)] // index access
```

Array

Access: first, last, count, isEmpty, min(), max()

Mutate: append(_:), insert(_:at:), remove(at:), removeFirst(),
removeLast(), removeAll(), swapAt(_:_:)

Order: sort(), sorted(), sorted(by: >), reverse(), reversed(),
shuffle()

Search: contains(_:), firstIndex(of:), first(where:),
contains(where:)

Slice: prefix(_:), suffix(_:), a[0..<3]

Init: Array(repeating:count:), Array(0..<10),
Array(stride(from:to:by:))

Dictionary

Access: d["k"], d["k", default: 0], d.keys, d.values, d.count,
d.isEmpty

Mutate: d["k"] = v, removeValue(forKey:), merge(_:uniquingKeysWith:),
mapValues(_:)

Query: contains(where:)

```
// Counting: d[key, default: 0] += 1
// Freq map: arr.reduce(into: [:]) { $0[$1, default: 0] += 1 }
```

Sparse Grid (avoids 2D array + bounds checking)

```
var grid = [[Int]: Int]() // [Int] is Hashable
grid[[23, 77]] = 1 // no allocation, no bounds
grid[[-1, -1], default: 0] // missing keys -> default
// Works for any dim: grid[[x, y, z]] for 3D
```

Set

Mutate: insert(_:), remove(_:), contains(_:), count, isEmpty

Algebra: union(_:), intersection(_:), subtracting(_:),
symmetricDifference(_:), isSubset(of:), isSuperset(of:)

Tuple & Comparable

```
let pair = (x: 1, y: 2) // pair.x or pair.0
// Comparable element-wise (up to 6 elements):
(1, "b") < (2, "a") // true -- compares 1st first
```

Higher-Order Functions

```
.map { $0 * 2 }
.flatMap { $0 }
.reduce(0, +)
.forEach { print($0) }
.enumerated()
.first(where: { $0 > 1 })
.prefix(while: { $0 < 3 })
zip(a, b)

.compactMap { Int($0) }
.filter { $0 % 2 == 0 }
.reduce(into: [:]) { ... }
.sorted(by: >)
.allSatisfy { $0 > 0 }
.contains(where: { $0 > 2 })
.drop(while: { $0 < 3 })
for (i, v) in arr.enumerated() {}
```

NeetCode DSA Categories

- Arrays & Hashing** freq maps, two sum, group anagrams, top-K frequent
- Two Pointers** valid palindrome, 3sum, container with most water
- Sliding Window** best time buy/sell, longest substr w/o repeating
- Stack** valid parens, min stack, eval reverse polish, daily temps
- Binary Search** search rotated arr, find min rotated, koko bananas
- Linked List** reverse, merge two sorted, detect cycle, LRU cache
- Trees** invert, max depth, level order, validate BST, serialize
- Tries** implement trie, word search II
- Heap / Priority Queue** kth largest, merge k sorted, find median stream
- Backtracking** subsets, permutations, combination sum, n-queens
- Graphs** num islands, clone graph, course schedule, word ladder
- Adv. Graphs** Dijkstra, Prim/Kruskal MST, network delay time
- 1-D DP** climbing stairs, house robber, LIS, coin change
- 2-D DP** unique paths, LCS, edit distance
- Greedy** max subarray (Kadane), jump game, gas station
- Intervals** merge intervals, insert interval, meeting rooms
- Math & Geometry** rotate image, spiral matrix, set matrix zeroes
- Bit Manipulation** single number, counting bits, reverse bits

DSA Patterns - Swift Idioms

Hash Map Counting

```
for c in str { freq[c, default: 0] += 1 }
```

Two Pointers

```
var lo = 0, hi = arr.count - 1
while lo < hi { lo += 1; hi -= 1 }
```

Sliding Window

```
var left = 0
for right in 0..
```

BFS (head-index: O(1) dequeue)

```
var queue = [start]; var head = 0
var visited = Set([start])
while head < queue.count {
    let node = queue[head]; head += 1
    for nb in graph[node, default: []]
        where !visited.contains(nb) {
            visited.insert(nb); queue.append(nb)
        }
}
// removeFirst() is O(n) -- head index is O(1)
```

DFS (Recursive)

```
func dfs(_ node: Int) {
    visited.insert(node)
    for nb in graph[node, default: []]
        where !visited.contains(nb) {
            dfs(nb)
        }
}
```

Binary Search

```
var lo = 0, hi = arr.count - 1
while lo <= hi {
    let mid = lo + (hi - lo) / 2
    if arr[mid] == target { return mid }
    arr[mid] < target ? (lo = mid + 1) : (hi = mid - 1)
}
```

Heap / Top-K

```
// No stdlib heap -- sort-based top-K:
arr.sorted(by: >).prefix(k)
// O(n log k): import Collections -> Heap<Int>
```

Stack / Monotonic Stack

```
// Stack: append (push), removeLast (pop), last (peek)
// Monotonic (next greater element):
for i in 0..
```

Deque (Sliding Window Max)

```
var deque = [Int]() // indices, front = max
for i in 0..
```

iOS APIs & Networking

UserDefaults

```
let ud = UserDefaults.standard
ud.set(value, forKey: "k")    // Int, String, Bool, Array, Data
ud.integer(forKey:)           // .string, .bool, .array, .double
ud.removeObject(forKey: "k")
// Codable structs won't auto-decode -- use Data + JSONEncoder/Decoder
```

FileManager

```
let fm = FileManager.default
let docs = fm.urls(for: .documentDirectory, in: .userDomainMask)[0]
let url = docs.appending(path: "data.json") // iOS 16+
try data.write(to: url)                     // write
let d = try Data(contentsOf: url)           // read
fm.fileExists(atPath: url.path)             // check
try fm.removeItem(at: url)                  // delete
try fm.contentsOfDirectory(at: docs, includingPropertiesForKeys: nil)
```

URLSession GET + JSONDecoder

```
let url = URL(string: "https://api.example.com/items")!
let (data, resp) = try await URLSession.shared.data(from: url)
guard let http = resp as? HTTPURLResponse,
      http.statusCode == 200
else { throw URLError(.badServerResponse) }
let items = try JSONDecoder().decode([Item].self, from: data)
```

With URLRequest & Decoder Options

```
var req = URLRequest(url: url)
req.setValue("application/json", forHTTPHeaderField: "Accept")
let (data, _) = try await URLSession.shared.data(for: req)
```

```
let dec = JSONDecoder()
dec.keyDecodingStrategy = .convertFromSnakeCase
dec.dateDecodingStrategy = .iso8601
```

Structured Concurrency

async let (parallel calls)

```
async let user = fetchUser(id: 1)
async let posts = fetchPosts(id: 1)
let (u, p) = try await (user, posts) // both run concurrently
```

TaskGroup (dynamic parallelism)

```
let results = await withTaskGroup(of: String.self) { group in
    for url in urls {
        group.addTask(name: "fetch-\(url)") {
            await fetch(url)
        }
    }
    var collected = [String]()
    for await result in group { collected.append(result) }
    return collected
}
// withThrowingTaskGroup for tasks that can throw
```

Task (Unstructured)

```
// Fire-and-forget (inherits actor context)
Task(name: "refreshData") { await refreshData() }

// Store handle -- access value or cancel later
let task = Task<String, Error>(name: "fetchName") {
    try await fetchUserName()
}
let name = try await task.value // blocks until done
task.cancel()                  // cooperative cancel

// Detached -- does NOT inherit actor context
Task.detached(name: "bg-export", priority: .background) {
    await exportLargeFile()
}

// Read current task name (useful in logging)
if let n = Task.name { print("Running: \(n)") }
```

```
// Priorities: .userInitiated .medium .utility .background
```

SwiftUI .task

```
// Runs on appear, auto-cancelled on disappear
.task { await viewModel.loadData() }
```

```
// Re-runs when id changes (old task cancelled)
.task(id: selectedTab) {
    await viewModel.load(tab: selectedTab)
}
```

SwiftUI Property Wrappers

Value Types (view-owned state)

```
@State private var count = 0 // View OWNS. Survives re-renders.
@Binding var isOn: Bool      // Two-way ref to parent @State.
// Created with $: Toggle(isOn: $vm.isOn)
```

Reference Types (ObservableObject)

```
@StateObject private var vm = VM() // View OWNS. Created once.
@ObservedObject var vm: VM         // Passed in. NOT owned.
@EnvironmentObject var vm: VM       // via .environmentObject()
```

Inside ObservableObject

```
class VM: ObservableObject {
    @Published var items: [Item] = [] // triggers view re-render
}
```

Environment & Persistence

```
@Environment(\.colorScheme) var scheme // system values
@Environment(\.dismiss) var dismiss
@AppStorage("username") var name = "" // UserDefaults-backed
@SceneStorage("draft") var draft = "" // per-scene state
@FocusState private var focused: Bool // keyboard focus
```

iOS 17+ @Observable (replaces ObservableObject)

```
@Observable class VM { // no @Published needed
    var items: [Item] = [] // all props tracked auto
}
@Bindable var vm: VM // creates bindings to props
```

```
@State/@StateObject: view OWNS (creation)
@Binding/@ObservedObject: view BORROWS (from parent)
@EnvironmentObject: view FINDS (injected up tree)
@Observable + @Bindable: iOS 17+ (simpler, efficient)
```

SwiftUI Components

Text & Labels

```
Text("Hello") // static/dynamic text
Label("Settings", systemImage: "gear") // icon + text
Image(systemName: "star.fill") // SF Symbol
Image("photo") // asset catalog
.resizable() // required to resize
.aspectRatio(contentMode: .fill) // .fit or .fill
.frame(width: 200, height: 150)
.clipShape(RoundedRectangle(cornerRadius: 12))
AsyncImage(url: url) // remote image
```

Buttons & Input

```
Button("Tap") { action() } // tappable
Toggle("Wi-Fi", isOn: $isOn) // on/off switch
Slider(value: $v, in: 0..100) // range
Stepper("Qty: \(q)", value: $q) // +/- buttons
Picker("Sort", selection: $s) { } // dropdown/segmented
DatePicker("Date", selection: $d) // date/time
ColorPicker("Color", selection: $c) // color
```

Text Input

```
TextField("Name", text: $text) // single-line
SecureField("Pass", text: $pw) // masked
TextEditor(text: $body) // multi-line
```

iOS APIs & Networking (cont.)

Layout

```
VStack { } HStack { } ZStack { } // stacks
LazyVStack { } LazyHStack { } // lazy (scroll perf)
Grid { GridRow { } } // 2D grid (iOS 16+)
Spacer() Divider() // fill / separator
GeometryReader { geo in } // parent size/pos
```

Lists & Scroll

```
List(items) { item in Row(item) } // scrollable rows
ForEach(items) { item in ... } // loop in containers
ScrollView(.vertical) { } // free-form scroll
LazyVGrid(columns: cols) { } // vertical grid
LazyHGrid(rows: rows) { } // horizontal grid
```

List Skeleton (UITableView analog)

```
List(selection: $selected) {
    Section("Favorites") { // header
        ForEach(filtered) { item in
            Row(item: item)
                .swipeActions(edge: .trailing) {
                    Button("Delete",
                        role: .destructive) { del(item) }
                }
                .swipeActions(edge: .leading) {
                    Button("Pin") { pin(item) }
                }
        }
        .onDelete { offsets in remove(offsets) }
        .onMove { from, to in move(from, to) }
    } footer: {
        Text("\(filtered.count) items")
    }
}

.searchable(text: $query) // search bar
.refreshable { await reload() } // pull-to-refresh
.listStyle(.insetGrouped) // .plain .grouped
.environment(\.editMode, $mode) // enable edit mode
```

List = UICollectionView under the hood (iOS 16+).
Cell reuse, lazy rendering, diffable updates, prefetch.
ForEach in List is lazy; in ScrollView/VStack is eager.

Navigation

```
NavigationStack { } // push/pop (iOS 16+)
NavigationLink("Detail") { View() } // push trigger
NavigationSplitView { } detail: { } // sidebar+detail
TabView { }.tabItem { Label(...) } // tab bar
```

NavigationDestination (data-driven)

```
NavigationLink(value: item) { Text(item.name) }
.navigationDestination(for: Item.self) { item in
    DetailView(item: item)
}
```

Coordinator Pattern (iOS 17+)

```
enum Route: Hashable {
    case detail(Item)
    case settings
}

@Observable class Coordinator {
    var path = NavigationPath()
    func push(_ r: Route) { path.append(r) }
    func pop() { path.removeLast() }
    func popToRoot() { path.removeLast(path.count) }

    @ViewBuilder
    func view(for route: Route) -> some View {
        switch route {
            case .detail(let item): DetailView(item: item)
            case .settings: SettingsView()
        }
    }
}
```

NavigationStack + Path

```
@State private var coordinator = Coordinator()

NavigationStack(path: $coordinator.path) {
    HomeView()
        .navigationDestination(for: Route.self) {
            coordinator.view(for: $0)
        }
}
```

Containers & Presentation

```
Form { } Section("Hdr") { } // settings list
GroupBox("Title") { } // bordered box
DisclosureGroup("More") { } // expandable
.sheet(isPresented: $show) { } // modal sheet
.alert("Title", isPresented: $s) // alert dialog
.fullScreenCover(...) // full-screen modal
.popover(isPresented: $show) { } // popover (iPad)
.confirmationDialog(...) // action sheet
```

Progress & Status

```
ProgressView() // spinner
ProgressView(value: 0.5) // progress bar
Gauge(value: 0.7) { Text("CPU") } // gauge (iOS 16+)
```

Maps & Web

```
Map(position: $pos) { } // MapKit (iOS 17+)
// WKWebView: use UIViewRepresentable
```

Codable Struct + CodingKeys

```
struct User: Codable {
    let id: Int
    let firstName: String
    let avatarURL: String
    let createdAt: Date

    enum CodingKeys: String, CodingKey {
        case id
        case firstName = "first_name"
        case avatarURL = "avatar_url"
        case createdAt = "created_at"
    }
}
```

CodingKeys map JSON snake_case to Swift camelCase.

Alt: `decoder.keyDecodingStrategy = .convertFromSnakeCase`

Custom Decoding

```
struct Post: Codable {
    let id: Int
    let title: String
    let tags: [String]

    enum CodingKeys: String, CodingKey {
        case id, title
        case tags = "post_tags" // JSON key differs
    }

    init(from decoder: Decoder) throws {
        let c = try decoder.container(keyedBy: CodingKeys.self)
        id = try c.decode(Int.self, forKey: .id)
        title = try c.decode(String.self, forKey: .title)
        tags = try c.decodeIfPresent([String].self,
            forKey: .tags) ?? []
    }
}
```

`decodeIfPresent`: nil if key missing/null (vs throwing)

`nestedContainer`: for nested JSON objects

`singleValueContainer`: for wrapper types

`unkeyedContainer`: for arrays element-by-element

Resilient Enum (DTO Best Practice)

Problem: API adds new enum values -> app crashes on decode

Solution: catch-all case preserves the raw value

iOS APIs & Networking (cont.)

```
enum Status: Decodable, Equatable {  
    case active, inactive, archived  
    case unknown(String)    // catch-all  
  
    init(from decoder: Decoder) throws {  
        let val = try decoder.singleValueContainer()  
            .decode(String.self)  
  
        switch val {  
        case "active":    self = .active  
        case "inactive": self = .inactive  
        case "archived": self = .archived  
        default:          self = .unknown(val)  
        }  
    }  
}
```

```
{"status":"pending"} -> .unknown("pending")
```

Platform Patterns

In-Flight Coalescing Cache

Dedup concurrent requests for the same key.
Common for: image loading, API calls, token refresh.

```
actor RequestCoalescer<Key: Hashable, Value: Sendable> {
    private var inFlight: [Key: Task<Value, Error>] = [:]

    func fetch(
        key: Key,
        work: @Sendable () async throws -> Value
    ) async throws -> Value {
        if let existing = inFlight[key] {
            return try await existing.value // reuse
        }
        let task = Task { try await work() }
        inFlight[key] = task
        defer { inFlight[key] = nil }
        return try await task.value
    }
}

// Usage -- Data is Sendable (UIImage is NOT)
let loader = RequestCoalescer<URL, Data>()
let data = try await loader.fetch(key: url) {
    let (d, _) = try await URLSession
        .shared.data(from: url)
    return d
}

let image = UIImage(data: data) // decode on caller
```

Actor Reentrancy

Each await is a suspension point -- other callers
can interleave. Never split a critical section
(read-modify-write) across an await.

```
// BUG: read-modify-write SPLIT across await
actor Cache {
    var items = [Item]()
    func process() async {
        let current = items // READ
        let new = await fetch() // SUSPEND!
        items = current + [new] // WRITE (stale!)
        // Call B can interleave at suspend --
        // both read same state, last write wins,
        // first caller's append is LOST
    }
}
```

Fix: keep critical section atomic

```
// GOOD: async work ABOVE, state update below
func process() async {
    let new = await fetch() // suspend here
    items.append(new) // atomic read+write
}

// GOOD: state update ABOVE, async work below
func enqueue(_ item: Item) async {
    items.append(item) // atomic read+write
    await sync() // suspend here
}
```

Rule: critical section above OR below the await,
never split across it. Each synchronous block on
an actor is serialized -- no interleaving.

MainActor Isolation

```
// Entire class on main thread (ViewModels)
@MainActor
class ProfileVM: ObservableObject {
    @Published var name = ""
    func load() async throws {
        // already on MainActor
        name = try await api.fetchName()
    }
}
```

```
// One-off main thread hop
await MainActor.run { self.label = "Done" }
```

```
// Mark a function
@MainActor func updateUI() { ... }
```

```
// nonisolated: opt out for pure computation
nonisolated func hash(into: inout Hasher) { }
```

Sendable

Types safe to pass across actor/concurrency boundaries.

```
// Value types are implicitly Sendable
struct Point: Sendable { var x, y: Double }

// Classes must be final + immutable
final class Token: Sendable {
    let value: String // let only, no var
}

// Actors are always Sendable
// @Sendable closures: no mutable captures
Task { @Sendable in
    await process(item) // item must be Sendable
}
```

@unchecked Sendable: escape hatch when you
manage thread safety yourself (e.g. with locks).

Cooperative Cancellation

```
// Check cancellation (throws if cancelled)
try Task.checkCancellation()
```

```
// Poll cancellation (non-throwing)
guard !Task.isCancelled else { return partial }
```

```
// withTaskCancellationHandler
let data = try await withTaskCancellationHandler {
    try await longDownload()
} onCancel: {
    urlSessionTask.cancel() // bridge to non-async
}
```

Cancellation is cooperative -- tasks must check.
URLSession, Task.sleep, etc. check automatically.

AsyncSequence & AsyncStream

```
// Consuming an AsyncSequence
for await msg in webSocket.messages {
    handle(msg)
}
```

```
// Creating a custom stream (replaces delegate)
let stream = AsyncStream<CLLocation> { cont in
    locMgr.onUpdate = { cont.yield($0) }
    locMgr.onDone = { cont.finish() }
    cont.onTermination = { _ in
        locMgr.stop()
    }
}
```

```
// AsyncThrowingStream for errors
let s = AsyncThrowingStream<Data, Error> { c in
    c.yield(data)
    c.finish(throwing: err) // or c.finish()
}
```

Use AsyncStream to bridge delegates, callbacks,
NotificationCenter, KVO into async/await.