iOS

Interview Questions and Answers

Core Concepts

This section focuses on fundamental principles and advanced concepts that an experienced developer should master.

1. Explain the difference between value types and reference types in Swift. How does copy-on-write optimization work?

Value Types vs Reference Types

Value types (structs, enums, tuples) are copied when assigned or passed, while **reference types** (classes) share a single instance through references.

- **Value types:** Each variable holds its own independent copy of the data
- **Reference types:** Multiple variables can reference the same instance
- **Memory:** Value types typically live on the stack, references on the heap

Copy-on-Write (CoW) Optimization

Swift's standard library collections (Array, Dictionary, Set) use CoW to optimize performance. The data is shared between copies until one copy is modified, at which point an actual copy is made.

```
var array1 = [1, 2, 3]
var array2 = array1  // No copy yet
array2.append(4)     // Copy happens here
// array1 = [1, 2, 3]
// array2 = [1, 2, 3, 4]
```

This provides value semantics with reference-type performance characteristics, avoiding unnecessary copying until mutation occurs.

2. What is the difference between weak, strong, and unowned references? When would you use each?

Reference Types in Swift

Strong references (default) increment the retain count and prevent deallocation. **Weak references** don't increment retain count and become nil when deallocated. **Unowned references** don't increment retain count but assume the referenced object always exists.

- **Strong:** Default behavior, creates ownership
- **Weak:** Optional reference, becomes nil automatically when object deallocates
- **Unowned:** Non-optional reference, crashes if accessed after deallocation

Usage Guidelines

```
class Parent {
  var child: Child?
}
class Child {
  weak var parent: Parent?  // Weak to break retain cycle
  unowned let owner: Parent // Unowned when lifetime guaranteed
}
```

Use weak for delegate patterns, closures capturing self, and parent-child relationships where the child may outlive the parent. **Use unowned** when you're certain the referenced object will outlive the reference (like a view controller owning a view). **Use strong** for ownership relationships.

3. Explain the iOS app lifecycle and the role of AppDelegate and SceneDelegate. How has this changed with SwiftUI?

Traditional App Lifecycle (AppDelegate)

AppDelegate manages application-level events:

- **application(_:didFinishLaunchingWithOptions:)** - App launch initialization
- **applicationDidBecomeActive(_:)** - App enters foreground
- **applicationWillResignActive(_:)** - About to enter background
- **applicationDidEnterBackground(_:)** - Now in background
- **applicationWillTerminate(_:)** - App termination

SceneDelegate (iOS 13+)

Introduced for multi-window support on iPad. Manages scene-specific lifecycle:

```
func scene(_ scene: UIScene,
  willConnectTo session: UISceneSession) {
  // Scene setup
}
func sceneDidBecomeActive(_ scene: UIScene)
func sceneWillResignActive(_ scene: UIScene)
```

SwiftUI App Lifecycle

SwiftUI uses **@main** with App protocol, replacing AppDelegate/SceneDelegate with scene modifiers and environment values. Lifecycle events handled via **.onAppear**, **.onDisappear**, and **@Environment(\.scenePhase)** for monitoring active/inactive/background states.

4. What are the different types of iOS concurrency mechanisms? Compare GCD, Operation Queues, and async/await.

Grand Central Dispatch (GCD)

Low-level C API for managing concurrent operations using dispatch queues:

```
DispatchQueue.global().async {
  // Background work
  DispatchQueue.main.async {
    // Update UI
  }
}
```

Operation Queues

Higher-level abstraction built on GCD, providing dependencies, cancellation, and KVO:

- Subclass NSOperation for complex tasks
- Add dependencies between operations
- Control max concurrent operations
- Cancel operations in flight

Async/Await (Swift 5.5+)

Modern structured concurrency with readable, sequential code:

```
func fetchData() async throws -> Data {
  let (data, _) = try await URLSession.shared
    .data(from: url)
  return data
}
```

Comparison: GCD is lightweight and fast but harder to manage. Operations provide more control and cancellation. Async/await offers best ergonomics, automatic thread management, and prevents common concurrency bugs through compiler enforcement.

5. Explain the MVVM architecture pattern in iOS. How does it differ from MVC, and what are the benefits for SwiftUI?

MVVM Architecture

Model-View-ViewModel separates presentation logic from view logic:

- **Model:** Data and business logic
- **View:** UI presentation (passive, declarative)
- **ViewModel:** Presentation logic, transforms model data for view

MVVM vs MVC

In **MVC**, the ViewController becomes bloated with both view and presentation logic. In **MVVM**, the ViewModel handles presentation logic, making ViewControllers/Views thinner and more testable.

```
class UserViewModel: ObservableObject {
  @Published var username: String = ""
  @Published var isValid: Bool = false

  func validate() {
    isValid = username.count >= 3
  }
}
```

Benefits for SwiftUI

SwiftUI's declarative nature and data binding (**@Published**, **@ObservedObject**) make MVVM natural. The View automatically updates when ViewModel properties change. This separation enables:

- Easier unit testing of business logic
- Reusable ViewModels across platforms
- Clear separation of concerns
- Better support for reactive programming

6. How does ARC (Automatic Reference Counting) work in Swift? What are retain cycles and how do you prevent them?

Automatic Reference Counting

ARC automatically manages memory by tracking strong references to class instances. When an instance's reference count reaches zero, it's deallocated.

- Each strong reference increments the retain count
- When references are removed, count decrements
- At count = 0, deinit is called and memory freed

Retain Cycles

A **retain cycle** occurs when two objects hold strong references to each other, preventing deallocation:

```
class Person {
  var apartment: Apartment?
}
class Apartment {
  var tenant: Person?  // Retain cycle!
}
```

Prevention Strategies

1. Weak references for optional relationships:

```
weak var delegate: MyDelegate?
```

2. Unowned references for non-optional relationships:

```
unowned let owner: Owner
```

3. Capture lists in closures:

```
someMethod { [weak self] in
  self?.doSomething()
}
```

Always use weak/unowned for delegates, parent-child relationships, and closure captures that reference self.

7. What is protocol-oriented programming in Swift? Provide examples of protocol extensions and their benefits.

Protocol-Oriented Programming

POP favors composition through protocols over inheritance. Swift's protocols can have default implementations via extensions, enabling code reuse without class hierarchies.

```
protocol Drawable {
  func draw()
}
extension Drawable {
  func draw() {
    print("Default drawing")
  }
}
```

Protocol Extensions Benefits

- **Value type support:** Works with structs and enums, not just classes
- **Multiple conformance:** Types can adopt multiple protocols
- **Retroactive modeling:** Add protocol conformance to existing types
- **Default implementations:** Reduce boilerplate code

Advanced Example

```
protocol Identifiable {
  var id: String { get }
}
extension Identifiable {
  var displayID: String {
    return "ID: \(id)"
  }
}
struct User: Identifiable {
  let id: String
}
```

This approach promotes composition over inheritance, reduces coupling, and works seamlessly with Swift's value types, making code more flexible and testable.

8. Explain the difference between @State, @Binding, @ObservedObject, @StateObject, and @EnvironmentObject in SwiftUI.

SwiftUI Property Wrappers

@State - Private, value-type state owned by the view:

```
@State private var count = 0
```

@Binding - Two-way connection to state owned elsewhere:

```
@Binding var isOn: Bool
```

@StateObject - Creates and owns a reference-type ObservableObject (iOS 14+):

```
@StateObject var viewModel = MyViewModel()
```

@ObservedObject - Observes an ObservableObject owned elsewhere:

```
@ObservedObject var viewModel: MyViewModel
```

@EnvironmentObject - Dependency injection for shared objects:

```
@EnvironmentObject var settings: AppSettings
```

Usage Guidelines

- **@State:** Simple view-local state (Bool, Int, String)
- **@Binding:** Pass state to child views for modification
- **@StateObject:** Create and own ViewModels in the view
- **@ObservedObject:** Observe ViewModels passed from parent
- **@EnvironmentObject:** App-wide shared state (user settings, auth)

Choose based on ownership and scope. StateObject owns, ObservedObject observes, State for values, Binding for passing write access.

9. How do you handle memory management with closures in Swift? Explain capture lists and when to use [weak self] vs [unowned self].

Closure Capture Semantics

Closures capture references to variables from their surrounding context. By default, these are **strong captures**, which can create retain cycles when closures reference self.

Capture Lists

Capture lists specify how values are captured:

```
someAsyncCall { [weak self] result in
  guard let self = self else { return }
  self.handleResult(result)
}
```

Weak vs Unowned

[weak self] - Creates optional reference, safe if self might be deallocated:

- Use for async operations that may outlive self
- Requires unwrapping (guard let or optional chaining)
- Safe default choice

[unowned self] - Creates non-optional reference, assumes self exists:

```
resource.onComplete { [unowned self] in
  self.updateUI()  // Crashes if self deallocated
}
```

- Use when closure lifetime is guaranteed shorter than self
- Slightly better performance (no optional)
- Dangerous if assumption violated

Best practice: Use [weak self] for most closures, especially async operations. Only use [unowned self] when you're absolutely certain self will outlive the closure.

10. What are Combine publishers, subscribers, and operators? How does Combine compare to RxSwift and async/await?

Combine Framework

Apple's reactive programming framework with three core concepts:

- **Publisher:** Emits values over time
- **Subscriber:** Receives values from publisher
- **Operator:** Transforms, filters, or combines publishers

```
let publisher = [1, 2, 3].publisher
publisher
  .map { $0 * 2 }
  .filter { $0 > 2 }
  .sink { print($0) }
  .store(in: &cancellables)
```

Common Operators

map, **filter**, **flatMap**, **combineLatest**, **debounce**, **removeDuplicates**, **catch** for error handling.

Comparison

Combine vs RxSwift: Similar concepts (Observable/Publisher), but Combine is native, type-safe, and integrated with SwiftUI. RxSwift has richer operator library and cross-platform support.**Combine vs async/await:** Combine handles streams of values over time. Async/await handles single async operations. Combine better for UI bindings and reactive streams; async/await better for sequential async operations. They complement each other - use **.values** to bridge Combine publishers to async sequences.

Data Structures and Algorithms

Questions in this section test your understanding of how to work with and manipulate data efficiently.

1. Implement a Stack using two Queues in Swift. What is the time complexity of push and pop operations?

Stack Using Two Queues

We can implement a stack using two queues by making either push or pop operation costly. Here's an implementation where **push is O(n)** and **pop is O(1)**:

```
class StackUsingQueues {
    private var queue1: [T] = []
    private var queue2: [T] = []

    func push(_ element: T) {
        queue2.append(element)
        while !queue1.isEmpty { queue2.append(queue1.removeFirst()) }
        swap(&queue1, &queue2)
    }
    func pop() -> T? { return queue1.isEmpty ? nil : queue1.removeFirst() }
}
```

Time Complexity:

- Push: O(n) - we transfer all elements from queue1 to queue2
- Pop: O(1) - direct removal from front of queue1
- Space Complexity: O(n) for storing n elements

2. How would you implement an LRU (Least Recently Used) Cache in Swift? Explain the data structures used and time complexity.

LRU Cache Implementation

An efficient LRU cache uses a **doubly linked list** combined with a **dictionary (hash map)** to achieve O(1) for both get and put operations:

```
class LRUCache {
    private var capacity: Int
    private var cache: [Key: Node] = [:]
    private var head: Node?
    private var tail: Node?

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

Key Design Points:

- Dictionary stores key-to-node mappings for O(1) lookup
- Doubly linked list maintains access order (most recent at head)
- Get operation: Move accessed node to head
- Put operation: Add to head, remove tail if capacity exceeded
- Both operations: **O(1) time complexity**

3. Given an array of integers, find all pairs that sum to a target value. What's the optimal approach and its complexity?

Two Sum - All Pairs

The optimal approach uses a **hash set** to achieve O(n) time complexity with a single pass:

```
func findPairs(_ nums: [Int], target: Int) -> [(Int, Int)] {
    var seen = Set()
    var pairs = [(Int, Int)]()
    for num in nums {
        let complement = target - num
        if seen.contains(complement) { pairs.append((complement, num)) }
        seen.insert(num)
    }
    return pairs
}
```

Complexity Analysis:

- Time: O(n) - single pass through array
- Space: O(n) - hash set stores up to n elements
- Alternative sorting approach: O(n log n) time, O(1) space with two pointers

Note: For unique pairs or avoiding duplicates, use a Set of tuples or sort pairs before insertion.

4. Implement a Trie (Prefix Tree) in Swift with insert, search, and startsWith methods. What are the time complexities?

Trie Implementation

A **Trie** is ideal for prefix-based searches and autocomplete features:

```
class TrieNode {
    var children: [Character: TrieNode] = [:]
    var isEndOfWord = false
}
class Trie {
    private let root = TrieNode()
    func insert(_ word: String) {
        var node = root
        for char in word { node = node.children[char, default: TrieNode()] }
        node.isEndOfWord = true
    }
}
```

Time Complexity:

- Insert: O(m) where m is word length
- Search: O(m) to traverse the word
- StartsWith: O(m) to check prefix
- Space: O(ALPHABET_SIZE * N * M) worst case

Use Cases: Autocomplete, spell checkers, IP routing tables

5. Explain the difference between Array, Set, and Dictionary in Swift. What are their underlying implementations and lookup complexities?

Swift Collection Types

Array:

- Ordered collection, allows duplicates
- Contiguous memory buffer (similar to dynamic array)
- Access by index: O(1), Search: O(n), Append: O(1) amortized

Set:

- Unordered, unique elements, conforms to Hashable
- Hash table implementation
- Insert/Delete/Contains: O(1) average, O(n) worst case

Dictionary:

- Key-value pairs, keys must be Hashable
- Hash table with separate chaining or open addressing
- Lookup/Insert/Delete: O(1) average, O(n) worst case

Memory: Arrays are most memory-efficient, Sets/Dictionaries have overhead for hash table buckets.

6. Implement a function to find the maximum sum of a contiguous subarray (Kadane's Algorithm). Explain the approach.

Kadane's Algorithm - Maximum Subarray

This classic **dynamic programming** problem can be solved in O(n) time with O(1) space:

```
func maxSubarraySum(_ nums: [Int]) -> Int {
    guard !nums.isEmpty else { return 0 }
    var maxSum = nums[0]
    var currentSum = nums[0]
    for i in 1..
```

Key Insight:

- At each position, decide: start new subarray or extend current one
- Track both current sum and global maximum
- Time: O(n), Space: O(1)
- Handles all negative numbers correctly

7. How do you detect a cycle in a linked list? Implement Floyd's Cycle Detection Algorithm and explain its complexity.

Floyd's Cycle Detection (Tortoise and Hare)

Uses **two pointers** moving at different speeds to detect cycles in O(n) time with O(1) space:

```
class ListNode {
    var val: Int
    var next: ListNode?
    init(_ val: Int) { self.val = val }
}
func hasCycle(_ head: ListNode?) -> Bool {
    var slow = head, fast = head
    while fast != nil && fast?.next != nil {
        slow = slow?.next; fast = fast?.next?.next
        if slow === fast { return true }
    }
    return false
}
```

Why It Works:

- If cycle exists, fast pointer will eventually catch slow pointer
- Time: O(n) - fast pointer traverses at most 2n nodes
- Space: O(1) - only two pointers used

8. Implement a sliding window algorithm to find the maximum sum of k consecutive elements in an array. What's the time complexity?

Sliding Window Maximum

The **sliding window technique** optimizes the solution from O(n*k) to O(n):

```
func maxSumSubarray(_ nums: [Int], _ k: Int) -> Int? {
    guard nums.count >= k else { return nil }
    var windowSum = nums[0..
```

Technique Breakdown:

- Calculate initial window sum
- Slide window: subtract left element, add right element
- Time: O(n) - single pass after initial window

- Space: O(1) - constant extra space
- Applicable to many problems: longest substring, minimum window, etc.

9. What is the difference between BFS and DFS for tree/graph traversal? When would you use each in iOS development?

BFS vs DFS Comparison

Breadth-First Search (BFS):

- Uses Queue (FIFO), explores level by level
- Space: O(w) where w is maximum width
- Finds shortest path in unweighted graphs
- iOS Use: View hierarchy traversal, finding nearest neighbor

Depth-First Search (DFS):

- Uses Stack (LIFO) or recursion, explores deep first
- Space: O(h) where h is height
- Better for path existence, topological sort
- iOS Use: File system traversal, dependency resolution, navigation stack

Time Complexity: Both O(V + E) for graphs, O(n) for trees

Practical Example: Use BFS for UIView subview searching by level, DFS for responder chain traversal.

10. Implement a binary search algorithm in Swift. What are the edge cases and how do you avoid integer overflow?

Binary Search Implementation

Binary search achieves **O(log n)** time complexity for sorted arrays:

```
func binarySearch(_ arr: [T], _ target: T) -> Int? {
    var left = 0, right = arr.count - 1
    while left <= right {
        let mid = left + (right - left) / 2
        if arr[mid] == target { return mid }
        else if arr[mid] < target { left = mid + 1 }
        else { right = mid - 1 }
    }
    return nil
}
```

Critical Points:

- Use left + (right - left) / 2 instead of (left + right) / 2 to avoid overflow
- Edge cases: empty array, single element, target not found
- Loop condition: left <= right (not just <)
- Time: O(log n), Space: O(1)

System Design

These questions evaluate your ability to think about the bigger picture, including architecture, scalability, and performance.

1. Design a scalable URL shortener service like bit.ly. What are the key components and how would you handle high traffic?

System Architecture

A URL shortener requires several key components:

- **API Gateway:** Routes requests to appropriate services
- **Application Servers:** Handle URL generation and retrieval
- **Database:** Store URL mappings (original URL, short code, metadata)
- **Cache Layer:** Redis/Memcached for frequently accessed URLs
- **CDN:** Serve redirects from edge locations

Key Design Decisions

Short URL Generation: Use base62 encoding (a-z, A-Z, 0-9) to generate 6-7 character codes. For 7 characters: 62^7 = 3.5 trillion possible URLs.

```
func generateShortCode(id: Int64) -> String {
    let base62 = "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
    var num = id
    var result = ""
    while num > 0 {
        result = String(base62[base62.index(base62.startIndex, offsetBy: Int(num % 62))]) + result
        num /= 62
    }
    return result
}
```

Database Schema: Use a distributed database with sharding based on short code hash. Consider NoSQL (Cassandra/DynamoDB) for horizontal scalability.

Caching Strategy: Cache hot URLs (80/20 rule) with TTL. Use write-through cache for new URLs.

Scalability: Stateless application servers behind load balancers, database replication with master-slave setup, and rate limiting per user/IP.

Analytics: Use message queues (Kafka) to asynchronously process click events without impacting redirect latency.

2. How would you design a real-time chat application for iOS? Discuss the architecture, data synchronization, and offline support.

Architecture Components

- **Client Layer:** iOS app with WebSocket connections
- **Gateway Layer:** WebSocket servers for real-time bidirectional communication
- **Service Layer:** Message processing, user management, presence detection
- **Storage Layer:** Message database (Cassandra/MongoDB), user database (PostgreSQL)
- **Cache Layer:** Redis for online users, recent messages
- **Message Queue:** Kafka/RabbitMQ for reliable message delivery

Real-Time Communication

Use **WebSocket** for persistent connections. Implement heartbeat mechanism to detect connection state:

```
class ChatWebSocketManager {
    private var webSocket: URLSessionWebSocketTask?

    func connect() {
        let session = URLSession(configuration: .default)
        webSocket = session.webSocketTask(with: URL(string: "wss://chat.api.com")!)
        webSocket?.resume()
        receiveMessage()
    }
}
```

Message Delivery Guarantees

At-least-once delivery: Each message gets a unique ID. Client acknowledges receipt. Server retries unacknowledged messages.

Offline Support

- Store messages locally using Core Data or Realm
- Queue outgoing messages when offline
- Sync on reconnection using last message timestamp
- Implement conflict resolution for concurrent edits

Scalability Considerations

Use **consistent hashing** to distribute users across WebSocket servers. Implement **pub-sub pattern** with Redis for message fanout to multiple recipients. Handle **presence** using Redis sorted sets with timestamps.

3. Design a news feed system like Instagram or Twitter. How would you handle feed generation, ranking, and real-time updates?

System Components

- **Feed Generation Service:** Creates personalized feeds
- **Ranking Service:** ML-based content scoring
- **Post Service:** Handles content creation/storage
- **Graph Service:** Manages social connections
- **Cache Layer:** Pre-computed feeds in Redis
- **CDN:** Media content delivery

Feed Generation Approaches

Fanout-on-Write (Push Model): When user posts, immediately write to all followers' feeds. Good for users with few followers.

Fanout-on-Read (Pull Model): Generate feed when user requests it by querying followees' recent posts. Good for celebrities with millions of followers.

Hybrid Approach: Use fanout-on-write for regular users, fanout-on-read for celebrities. Combine at read time.

```
func generateFeed(userId: String) async -> [Post] {
    let cachedFeed = await cache.get("feed:\(userId)")
    if let feed = cachedFeed { return feed }

    let followees = await graphService.getFollowees(userId)
    let posts = await postService.getRecentPosts(followees)
    let rankedPosts = await rankingService.rank(posts, for: userId)
    await cache.set("feed:\(userId)", rankedPosts, ttl: 300)
    return rankedPosts
}
```

Ranking Algorithm

Score posts based on: recency, engagement (likes/comments), user affinity, content type preference. Use ML models trained on user behavior.

Real-Time Updates

Use WebSocket or Server-Sent Events for new post notifications. Implement **incremental feed updates** rather than full refresh.

4. How would you design a video streaming service like Netflix for iOS? Discuss CDN strategy, adaptive bitrate streaming, and offline downloads.

High-Level Architecture

- **Client App:** iOS app with AVFoundation/AVPlayer
- **API Gateway:** Authentication, catalog, recommendations
- **Content Delivery Network:** Global edge servers for video delivery
- **Origin Servers:** Master video storage
- **Transcoding Service:** Convert videos to multiple formats/bitrates
- **Recommendation Engine:** ML-based content suggestions

Adaptive Bitrate Streaming

Use **HLS (HTTP Live Streaming)** protocol. Transcode videos into multiple bitrates (240p, 360p, 720p, 1080p, 4K). Client automatically switches based on network conditions.

```
let asset = AVURLAsset(url: hlsURL)
let playerItem = AVPlayerItem(asset: asset)
let player = AVPlayer(playerItem: playerItem)

playerItem.preferredPeakBitRate = 2_000_000
player.automaticallyWaitsToMinimizeStalling = true
```

CDN Strategy

Deploy content to **edge locations** near users. Use geographic DNS routing. Implement **cache warming** for popular content. Monitor CDN performance and failover to backup CDNs.

Offline Downloads

Use **AVAssetDownloadTask** for HLS downloads. Store encrypted content with FairPlay DRM. Implement download queue management, pause/resume capability, and storage limits.

Optimization Techniques

- Preload next episode during credits
- Use thumbnail sprites for seek preview
- Implement bandwidth prediction algorithms
- Support background audio for music content

5. Design a ride-sharing application like Uber. Focus on real-time location tracking, driver-rider matching, and handling high concurrency.

System Architecture

- **Location Service:** Track driver/rider positions in real-time
- **Matching Service:** Connect riders with nearby drivers
- **Trip Service:** Manage trip lifecycle
- **Pricing Service:** Calculate dynamic pricing
- **Payment Service:** Handle transactions
- **Notification Service:** Push notifications via APNs

Real-Time Location Tracking

Drivers send location updates every 4-5 seconds. Use **geospatial indexing** (Redis Geo, MongoDB geospatial queries, or dedicated solutions like Elasticsearch).

```
func startLocationTracking() {
    locationManager.allowsBackgroundLocationUpdates = true
    locationManager.startUpdatingLocation()
}
```

```
func locationManager(_ manager: CLLocationManager, didUpdateLocations locations: [CLLocation]) {
    guard let location = locations.last else { return }
    sendLocationToServer(lat: location.coordinate.latitude, lng: location.coordinate.longitude)
}
```

Driver-Rider Matching

Use **geohashing** to partition map into cells. Query drivers within same/adjacent cells. Rank by: distance, rating, acceptance rate. Implement **dispatch timeout** (15 seconds) before trying next driver.

Handling Concurrency

Use **distributed locks** (Redis) to prevent double-booking. Implement **optimistic locking** with version numbers in database. Use message queues for asynchronous processing of ride requests.

Scalability

Shard location data by geographic region. Use **WebSocket** connections for real-time updates. Implement rate limiting on location updates. Use **event sourcing** for trip state management.

6. How would you design a distributed cache system? Discuss consistency models, eviction policies, and handling cache invalidation.

Core Components

- **Cache Nodes:** In-memory data stores (Redis cluster)
- **Client Library:** Handles routing and failover
- **Configuration Service:** Manages node topology (ZooKeeper/Consul)
- **Monitoring:** Track hit rates, latency, memory usage

Data Distribution

Use **consistent hashing** to distribute keys across nodes. When nodes are added/removed, only K/n keys need redistribution (K=total keys, n=nodes). Virtual nodes improve load distribution.

```
class ConsistentHash {
    private var ring: [Int: String] = [:]
    private let virtualNodes = 150

    func getNode(for key: String) -> String {
        let hash = key.hashValue
        let sortedHashes = ring.keys.sorted()
        let node = sortedHashes.first { $0 >= hash } ?? sortedHashes.first!
        return ring[node]!
    }
}
```

Consistency Models

Strong Consistency: Read-after-write guarantee. Use synchronous replication. Higher latency.

Eventual Consistency: Async replication. Lower latency. May serve stale data briefly.

CAP Theorem: In distributed cache, typically choose AP (Availability + Partition tolerance) over consistency.

Eviction Policies

- **LRU (Least Recently Used):** Most common, good general purpose
- **LFU (Least Frequently Used):** Better for workloads with hot keys
- **TTL-based:** Expire keys after time period

Cache Invalidation Strategies

Write-through: Write to cache and DB simultaneously. **Write-back:** Write to cache, async to DB. **Cache-aside:** App manages cache population.

7. Design a notification system that can send push notifications, emails, and SMS at scale. How would you ensure delivery and handle rate limiting?

System Architecture

- **API Gateway:** Receives notification requests
- **Notification Service:** Orchestrates delivery across channels
- **Channel Handlers:** APNs (iOS), FCM (Android), Email (SendGrid), SMS (Twilio)
- **Message Queue:** Kafka/RabbitMQ for buffering
- **Template Service:** Manage notification templates
- **User Preference Service:** Store delivery preferences
- **Analytics Service:** Track delivery metrics

Message Flow

Request → Validation → Template Rendering → Priority Queue → Channel Selection → Delivery → Retry Logic → Analytics

```
struct Notification {
    let userId: String
    let channels: [Channel]
    let priority: Priority
    let payload: [String: Any]
}

enum Channel { case push, email, sms }
enum Priority { case high, normal, low }
```

Ensuring Delivery

- **Retry Mechanism:** Exponential backoff for failed deliveries
- **Dead Letter Queue:** Store permanently failed messages
- **Idempotency:** Use unique message IDs to prevent duplicates
- **Delivery Receipts:** Track acknowledgments from providers

Rate Limiting

Implement **token bucket algorithm** per user and globally. For APNs, respect Apple's rate limits. Use **priority queues** to process high-priority notifications first.

Scalability

Partition message queue by user ID or region. Use **batch processing** for bulk notifications. Implement **circuit breakers** for external service failures. Cache user preferences and device tokens.

8. How would you design a search engine for an e-commerce platform? Discuss indexing, ranking, and handling typos/synonyms.

System Components

- **Indexing Service:** Processes product catalog into search index
- **Search Service:** Handles query processing and ranking
- **Suggestion Service:** Autocomplete and spelling correction
- **Analytics Service:** Track search metrics and user behavior
- **Storage:** Elasticsearch/Solr for inverted index

Indexing Strategy

Build **inverted index** mapping terms to product IDs. Index product fields: title, description, category, brand, attributes. Use **n-grams** for partial matching.

```
struct ProductDocument {
    let id: String
    let title: String
    let description: String
    let category: String
    let price: Double
    let rating: Double
    let tags: [String]
}
```

Query Processing

Tokenization: Split query into terms. **Normalization:** Lowercase, remove stopwords. **Stemming:** Reduce words to root form ("running" → "run"). **Synonym expansion:** Include related terms.

Ranking Algorithm

Score based on: **TF-IDF** (term frequency-inverse document frequency), **BM25** algorithm, product popularity, user personalization, recency, price relevance.

Handling Typos

Use **fuzzy matching** with Levenshtein distance. Implement **phonetic algorithms** (Soundex, Metaphone). Build **spelling correction** using query logs and product catalog.

Performance Optimization

- Cache popular queries
- Use sharding for large catalogs
- Implement query rewriting for better results
- A/B test ranking algorithms

9. Design a rate limiter service that can be used across multiple microservices. What algorithms would you use and how would you handle distributed rate limiting?

Rate Limiting Algorithms

1. Token Bucket: Tokens added at fixed rate. Request consumes token. Allows burst traffic up to bucket capacity.

2. Leaky Bucket: Requests processed at fixed rate. Excess requests queued or dropped. Smooths traffic spikes.

3. Fixed Window: Count requests per time window. Simple but can allow 2x rate at window boundaries.

4. Sliding Window Log: Track timestamp of each request. Accurate but memory-intensive.

5. Sliding Window Counter: Hybrid approach with weighted counts from current and previous windows.

```
class TokenBucket {
    private var tokens: Double
    private let capacity: Double
    private let refillRate: Double
    private var lastRefill: Date

    func allowRequest() -> Bool {
        refillTokens()
        if tokens >= 1 {
            tokens -= 1
            return true
        }
        return false
    }
}
```

Distributed Rate Limiting

Use **Redis** as centralized store. Implement using Lua scripts for atomic operations. Key format: "ratelimit:user_id:endpoint"

Implementation Strategies

- **Per-User Limits:** Track by user ID or API key

- **Per-IP Limits:** Prevent abuse from anonymous users
- **Per-Endpoint Limits:** Different limits for different APIs
- **Hierarchical Limits:** Tenant-level and user-level limits

Response Handling

Return HTTP 429 with headers: X-RateLimit-Limit, X-RateLimit-Remaining, X-RateLimit-Reset, Retry-After

10. How would you design a file storage and sharing system like Dropbox? Discuss synchronization, conflict resolution, and handling large files.

System Architecture

- **Client Application:** iOS app with local file system monitoring
- **API Servers:** Handle metadata operations
- **Block Servers:** Store file chunks
- **Metadata Database:** File hierarchy, versions, sharing permissions
- **Object Storage:** S3/Azure Blob for actual file data
- **Notification Service:** Real-time sync notifications
- **CDN:** Fast file downloads globally

File Chunking Strategy

Split files into 4MB blocks. Use **content-defined chunking** with rolling hash (Rabin fingerprinting) to detect changed blocks. Only upload modified chunks.

```
func chunkFile(fileURL: URL) -> [Data] {
    let chunkSize = 4 * 1024 * 1024
    let fileData = try! Data(contentsOf: fileURL)
    var chunks: [Data] = []
    var offset = 0
    while offset < fileData.count {
        let length = min(chunkSize, fileData.count - offset)
        chunks.append(fileData.subdata(in: offset..
```

Synchronization

Use **version vectors** or **Merkle trees** to detect changes. Implement **delta sync** to transfer only differences. Use WebSocket for real-time notifications of remote changes.

Conflict Resolution

Detect conflicts using version numbers. Strategy: **Last-Write-Wins** with conflict copies ("file (conflicted copy).txt"). Allow manual merge for important files.

Optimization

- Deduplication using content hashing (SHA-256)
- Compression before upload
- Resume interrupted uploads
- Prioritize small files over large ones

Coding and Debugging

This section presents practical coding challenges and questions about debugging techniques.

1. Write a Swift function to flatten a nested array of integers.

Solution

Here's an efficient recursive approach to flatten a nested array:

```swift
func flatten(_ array: [Any]) -> [Int] {
    var result = [Int]()
    for element in array {
        if let intValue = element as? Int {
            result.append(intValue)
        } else if let nestedArray = element as? [Any] {
            result.append(contentsOf: flatten(nestedArray))
        }
    }
    return result
}
// Usage: flatten([1, [2, 3], [[4], 5]]) returns [1, 2, 3, 4, 5]
```

Key points:

- Uses type casting to handle both Int and nested arrays
- Recursive approach handles arbitrary nesting depth
- Time complexity: O(n) where n is total number of elements

2. How would you reverse a string in Swift while maintaining Unicode correctness?

Solution

When reversing strings, it's crucial to handle Unicode grapheme clusters correctly:

```swift
func reverseString(_ str: String) -> String {
    return String(str.reversed())
}

// For Unicode-safe reversal:
func reverseStringUnicodeSafe(_ str: String) -> String {
    let graphemes = str.unicodeScalars.reversed()
    return String(String.UnicodeScalarView(graphemes))
}
```

Important considerations:

- Swift's **reversed()** on String works with grapheme clusters by default
- Handles emojis and combining characters correctly (e.g., "□□□□")
- For performance with large strings, use indices instead of creating new strings
- Avoid using UTF-16 or UTF-8 views for reversal as they break multi-byte characters

3. Write a function to check if a string is a palindrome, ignoring spaces and case.

Solution

Here's an optimized two-pointer approach:

```swift
func isPalindrome(_ str: String) -> Bool {
    let cleaned = str.lowercased().filter { $0.isLetter || $0.isNumber }
    let chars = Array(cleaned)
    var left = 0, right = chars.count - 1
    while left < right {
        if chars[left] != chars[right] { return false }
        left += 1; right -= 1
    }
    return true
}
```

Key optimizations:

- Filters non-alphanumeric characters in a single pass
- Two-pointer technique avoids string reversal (O(n) time, O(n) space)
- Handles edge cases: empty strings, single characters
- For memory optimization, could use string indices instead of Array conversion

4. What debugging tools does Xcode provide for memory profiling and how do you use them?

Primary Memory Profiling Tools

1. Instruments - Leaks & Allocations:

- **Leaks instrument**: Detects memory that's allocated but no longer referenced
- **Allocations instrument**: Tracks all memory allocations, shows growth patterns
- Use mark generation to snapshot memory state between actions
- Filter by allocation type to find specific leaks

2. Memory Graph Debugger:

- Access via Debug Navigator or Debug menu during runtime
- Shows visual graph of object relationships and retain cycles
- Use **!** icon to identify leaked objects
- Right-click objects to see backtrace of allocation

3. Debug Memory Graph command:

- Use **malloc_history** to trace allocation history
- **leaks** command in terminal for command-line analysis
- Enable Malloc Stack logging in scheme settings for detailed traces

5. Explain how to handle exceptions and errors in Swift. What's the difference between try, try?, and try!?

Swift Error Handling

Error handling keywords:

- **try**: Must be used with do-catch block, allows handling specific errors
- **try?**: Converts result to optional, returns nil on error (silent failure)
- **try!**: Force-unwraps result, crashes app if error occurs (use only when certain)

```
// try with do-catch
do {
    let data = try loadData()
} catch DataError.notFound {
    print("File not found")
} catch {
    print("Error: \(error)")
}

// try? returns Optional
let data = try? loadData()  // nil if error

// try! force unwrap
let data = try! loadData()  // crashes if error
```

Best practices:

- Use **try-catch** for recoverable errors with specific handling
- Use **try?** when error details don't matter and nil is acceptable
- Avoid **try!** except in tests or when failure is truly impossible

- Define custom Error enums conforming to Error protocol

6. How would you detect and fix a retain cycle in iOS? Provide a code example.

Detecting and Fixing Retain Cycles

Detection methods:

- Use Memory Graph Debugger (shows circular references with ! icon)
- Instruments Leaks tool during runtime
- Enable malloc stack logging for allocation backtraces

Common retain cycle example and fix:

```
// PROBLEM: Strong reference cycle
class ViewController: UIViewController {
    var closure: (() -> Void)?

    func setupClosure() {
        closure = {
            self.view.backgroundColor = .red  // Captures self strongly
        }
    }
}

// SOLUTION: Use capture list with weak/unowned
func setupClosure() {
    closure = { [weak self] in
        self?.view.backgroundColor = .red
    }
}
```

Key strategies:

- Use **[weak self]** in closures when self might be deallocated
- Use **[unowned self]** when self is guaranteed to exist (use cautiously)
- Mark delegate properties as **weak**
- Break cycles in deinit if using manual reference management

7. Write a function to find the first non-repeating character in a string.

Solution

Efficient approach using a dictionary for character frequency:

```
func firstNonRepeating(_ str: String) -> Character? {
    var charCount = [Character: Int]()

    for char in str {
        charCount[char, default: 0] += 1
    }

    for char in str {
        if charCount[char] == 1 { return char }
    }
    return nil
}
```

Algorithm analysis:

- Time complexity: O(n) - two passes through string
- Space complexity: O(k) where k is unique characters
- First loop builds frequency map
- Second loop maintains original order to find first non-repeating
- Returns nil if all characters repeat

Alternative: For very large strings with limited character sets, could use an array of size 256 for ASCII.

8. What is method swizzling in iOS and when would you use it? Provide an example.

Method Swizzling Overview

Method swizzling is an Objective-C runtime technique that allows changing the implementation of existing methods at runtime by exchanging method implementations.

Use cases:

- Adding analytics/logging to existing methods without subclassing
- Debugging third-party frameworks
- Implementing aspect-oriented programming
- Hot-patching critical bugs (use with extreme caution)

```
extension UIViewController {
    static func swizzleViewDidAppear() {
        let original = #selector(viewDidAppear(_:))
        let swizzled = #selector(swizzled_viewDidAppear(_:))
        guard let originalMethod = class_getInstanceMethod(UIViewController.self, original),
              let swizzledMethod = class_getInstanceMethod(UIViewController.self, swizzled) else { return }
        method_exchangeImplementations(originalMethod, swizzledMethod)
    }

    @objc func swizzled_viewDidAppear(_ animated: Bool) {
        swizzled_viewDidAppear(animated)  // Calls original
        print("Analytics: \(type(of: self)) appeared")
    }
}
```

Important warnings:

- Can cause hard-to-debug issues if not done carefully
- Should be performed in +load or dispatch_once
- Always call the original implementation
- Consider using composition or protocols as safer alternatives

9. How do you debug crashes that only occur in release builds but not in debug builds?

Release-Only Crash Debugging Strategy

Common causes:

- Compiler optimizations exposing race conditions or undefined behavior
- Different memory management in release mode
- Assertions and debug checks disabled
- Bitcode or dead code stripping issues

Debugging techniques:

- **1. Enable debug symbols:** Build Settings → Debug Information Format → DWARF with dSYM
- **2. Disable optimizations temporarily:** Set Optimization Level to -O0 for release
- **3. Use crash reports:** Symbolicate crash logs from TestFlight/App Store
- **4. Enable sanitizers:** Thread Sanitizer, Address Sanitizer in scheme settings
- **5. Check for race conditions:** Use Thread Sanitizer and audit concurrent code
- **6. Review compiler warnings:** Often indicate undefined behavior

Specific checks:

- Verify all optionals are properly unwrapped
- Check for uninitialized variables
- Review force-unwraps and force-casts
- Audit C/Objective-C bridging code
- Test with Zombies enabled to catch over-released objects

10. Write a function to implement a simple LRU (Least Recently Used) cache in Swift.

LRU Cache Implementation

Using a combination of dictionary and doubly-linked list for O(1) operations:

```
class LRUCache {
    private var capacity: Int
    private var cache = [Key: Value]()
    private var order = [Key]()

    init(capacity: Int) { self.capacity = capacity }

    func get(_ key: Key) -> Value? {
        guard let value = cache[key] else { return nil }
        order.removeAll { $0 == key }
        order.append(key)
        return value
    }

    func put(_ key: Key, _ value: Value) {
        if cache[key] != nil { order.removeAll { $0 == key } }
        else if cache.count >= capacity, let lru = order.first {
            cache.removeValue(forKey: lru)
            order.removeFirst()
        }
        cache[key] = value
        order.append(key)
    }
}
```

Note: This simplified version uses array for clarity. For production with large capacity, implement a proper doubly-linked list to achieve true O(1) performance for all operations.

Key features:

- Evicts least recently used item when capacity reached
- Updates access order on get operations
- Thread-safety would require adding locks or using serial queue

Behavioral Questions

These questions assess your soft skills, problem-solving approach, and how you work in a team.

1. Tell me about a time when you had to optimize an iOS app's performance. What was your approach?

Situation: Our e-commerce app was experiencing slow scrolling in the product listing screen, with frame drops below 30 FPS, leading to user complaints and negative reviews.

Task: I was assigned to identify bottlenecks and improve the scrolling performance to achieve a consistent 60 FPS.

Action: I used Instruments (Time Profiler and Core Animation) to identify that image loading and cell configuration were blocking the main thread. I implemented asynchronous image loading with SDWebImage, added image caching, prefetched data using UICollectionViewDataSourcePrefetching, and moved heavy computations to background queues. I also reduced the view hierarchy complexity in custom cells.

Result: Scrolling performance improved to a consistent 60 FPS, the app's rating increased from 3.8 to 4.5 stars, and crash reports related to memory issues decreased by 40%.

2. Describe a situation where you had to resolve a critical production bug under tight deadlines.

Situation: A critical bug in our banking app caused transactions to fail for users on iOS 15, affecting approximately 30% of our user base. The issue was discovered on a Friday evening.

Task: I needed to quickly identify the root cause, implement a fix, and coordinate an emergency release over the weekend.

Action: I immediately set up remote debugging with crash logs and Firebase Analytics. I discovered that a recent URLSession configuration change was incompatible with iOS 15's networking stack. I reverted the problematic code, added proper error handling, implemented comprehensive unit tests for the networking layer, and worked with QA to expedite testing. I also added feature flags to prevent similar issues in the future.

Result: The hotfix was deployed within 12 hours, transaction success rates returned to 99.8%, and we implemented a new protocol requiring backward compatibility testing for all networking changes.

3. Tell me about a time when you disagreed with a technical decision made by your team or manager.

Situation: My team lead proposed rewriting our entire app in SwiftUI to modernize the codebase, despite having a stable UIKit app with 2 years of development.

Task: I needed to present my concerns about the rewrite while being respectful and constructive.

Action: I prepared a detailed analysis comparing both approaches, including migration effort estimates, team learning curve, backward compatibility concerns for iOS 13 users (20% of our base), and potential risks. I proposed an incremental approach: using SwiftUI for new features while maintaining the UIKit foundation, with gradual migration of isolated screens. I presented data showing this would deliver value faster and reduce risk.

Result: The team agreed to the hybrid approach. We successfully launched 3 new SwiftUI features within 2 months while maintaining stability. Six months later, we had migrated 40% of the app with zero production incidents related to the migration.

4. Describe a situation where you had to mentor a junior developer who was struggling with iOS concepts.

Situation: A junior developer on my team was struggling with memory management concepts, particularly retain cycles and weak/unowned references, causing multiple memory leaks in their code.

Task: I needed to help them understand these concepts thoroughly while maintaining their confidence and motivation.

Action: I scheduled weekly one-on-one sessions where I explained ARC fundamentals using visual diagrams and real-world analogies. I created a sample project demonstrating common retain cycle scenarios (delegate patterns, closures, and observers) and their solutions. I conducted pair programming sessions, encouraged them to use Instruments' Memory Graph Debugger, and reviewed their code with detailed feedback. I also shared curated articles and WWDC sessions.

Result: Within 6 weeks, the developer independently identified and fixed memory leaks in their code. They later presented a team knowledge-sharing session on memory management best practices, and memory-related crashes in our app decreased by 25%.

5. Tell me about a time when you had to balance technical debt with feature development.

Situation: Our app had accumulated significant technical debt in the networking layer, using outdated patterns and making the codebase difficult to maintain. Meanwhile, product management was pushing for new features to meet quarterly goals.

Task: I needed to advocate for addressing technical debt while not blocking critical business features.

Action: I documented specific pain points: 30% of bugs originated from the networking layer, onboarding new developers took 2 weeks longer due to complexity, and feature development velocity had decreased by 20%. I proposed a compromise: allocate 20% of each sprint to refactoring while delivering features. I created a phased refactoring plan, starting with the most problematic areas, and demonstrated quick wins by refactoring one module that reduced related bugs by 60%.

Result: Management approved the 80/20 split. Over 4 months, we refactored the networking layer using modern async/await patterns, reduced networking-related bugs by 70%, and actually increased feature delivery speed by 15% due to improved code maintainability.

6. Describe a time when you had to learn a new iOS technology or framework quickly to meet project requirements.

Situation: Our company decided to add AR features to our interior design app using ARKit, but I had no prior experience with augmented reality development, and the feature was scheduled for release in 6 weeks.

Task: I needed to become proficient in ARKit, understand 3D concepts, and deliver a production-ready feature within the tight timeline.

Action: I created a structured learning plan: spent the first week on Apple's ARKit documentation and WWDC sessions, built 3 prototype apps to understand plane detection, object placement, and lighting. I joined AR development communities on Slack and Discord for quick problem-solving, consulted with a 3D graphics expert for optimization techniques, and applied TDD practices to ensure code quality despite the learning curve.

Result: I delivered the AR furniture placement feature on time with smooth performance (60 FPS). The feature became our app's key differentiator, increasing user engagement by 45%. I later conducted a workshop to share ARKit knowledge with the team.

7. Tell me about a time when you improved the development workflow or processes for your iOS team.

Situation: Our iOS team's build times had grown to 15+ minutes for clean builds, significantly impacting productivity. Developers were context-switching during builds, and CI/CD pipelines were taking over 30 minutes.

Task: I volunteered to analyze and improve our build performance and overall development workflow.

Action: I conducted a comprehensive audit using Xcode's build timeline and identified issues: unnecessary dependencies, lack of modularization, and inefficient build settings. I proposed and implemented a modularization strategy, breaking the monolithic app into feature modules using

SPM. I configured incremental builds, enabled compilation caching, optimized CocoaPods integration, and set up distributed caching with a build cache server. I also created documentation and migration guides for the team.

Result: Clean build times decreased to 4 minutes (73% improvement), incremental builds to under 30 seconds, and CI/CD pipeline time reduced to 12 minutes. Developer satisfaction scores increased, and we estimated saving 10+ hours per developer per week.

8. Describe a situation where you had to handle conflicting priorities from multiple stakeholders.

Situation: I was simultaneously receiving urgent requests from three stakeholders: Product wanted a new social sharing feature, Customer Support needed fixes for top user-reported bugs, and the CTO wanted security improvements for an upcoming audit.

Task: I needed to prioritize effectively while managing expectations and maintaining relationships with all stakeholders.

Action: I scheduled a meeting with all stakeholders to discuss priorities transparently. I created a prioritization matrix evaluating each request by impact, urgency, effort, and risk. I presented data: security issues could result in App Store removal, top bugs affected 15% of users daily, and the social feature had uncertain ROI. I proposed a phased approach: immediate security fixes (1 week), critical bug fixes (1 week), then social features (2 weeks). I provided weekly progress updates to all stakeholders.

Result: All stakeholders agreed to the prioritization. We passed the security audit without issues, resolved bugs affecting 50,000+ users, and delivered the social feature with only a 2-week delay. Stakeholders appreciated the transparent communication approach.

9. Tell me about a time when you had to make a technical decision with incomplete information.

Situation: We needed to choose between implementing our payment processing with Stripe or building a custom solution, but we had limited information about future business requirements and international expansion plans, which would significantly impact the decision.

Task: I needed to make a recommendation that would be flexible enough to accommodate uncertain future needs while delivering immediate value.

Action: I conducted a risk analysis for both approaches, identifying key unknowns and their potential impact. I created a decision framework based on known constraints: 3-month timeline, team expertise, compliance requirements, and maintenance costs. I implemented a proof-of-concept for both solutions over 3 days to evaluate integration complexity. I designed an abstraction layer that would allow switching payment providers with minimal code changes, reducing the risk of the decision. I documented assumptions and recommended Stripe with the abstraction layer.

Result: We launched with Stripe in 2.5 months. When the company expanded to Asia 8 months later requiring a regional payment provider, the abstraction layer allowed us to integrate the new provider in just 1 week instead of an estimated 6 weeks.

10. Describe a time when you received critical feedback about your code or technical approach. How did you handle it?

Situation: During a code review, a senior architect pointed out that my implementation of a complex feature using Combine was over-engineered, difficult to test, and introduced unnecessary complexity for the team.

Task: I needed to objectively evaluate the feedback, set aside my ego, and determine the best path forward.

Action: I scheduled a follow-up discussion to understand the concerns in detail. I acknowledged that I had prioritized showcasing modern patterns over pragmatism and team readability. I asked for specific examples of simpler approaches and reviewed them with an open mind. I refactored the implementation using a more straightforward approach with traditional delegation patterns, which reduced code by 40% and improved testability. I also reflected on why I had over-engineered the solution and shared my learnings in a team retrospective.

Result: The refactored code was approved and became easier for the team to maintain. I learned to balance technical sophistication with practical team needs. The senior architect later mentioned that my receptiveness to feedback and quick action demonstrated strong professional maturity, which contributed to my promotion 4 months later.