Complete DSA Patterns & System Design Guide for Swift Interviews

Core DSA Patterns for Coding Interviews

1. Two Pointers Pattern

Use Cases: Array problems, string manipulation, palindromes

swift

```swift
func twoSum(_ nums: [Int], _ target: Int) -> [Int] {
    var left = 0, right = nums.count - 1
    let sortedNums = nums.enumerated().sorted { $0.element < $1.element }

    while left < right {
        let sum = sortedNums[left].element + sortedNums[right].element
        if sum == target {
            return [sortedNums[left].offset, sortedNums[right].offset]
        } else if sum < target {
            left += 1
        } else {
            right -= 1
        }
    }
    return []
}
```

2. Fast & Slow Pointers (Floyd's Cycle Detection)

Use Cases: Linked list cycles, finding middle element

swift

```swift
class ListNode {
    var val: Int
    var next: ListNode?
    init(_ val: Int) { self.val = val }
}

func hasCycle(_ head: ListNode?) -> Bool {
    var slow = head
    var fast = head

    while fast?.next != nil {
        slow = slow?.next
        fast = fast?.next?.next
        if slow === fast { return true }
    }
    return false
}
```

3. Sliding Window Pattern

Use Cases: Substring problems, maximum/minimum subarray

swift

```swift
func lengthOfLongestSubstring(_ s: String) -> Int {
    var charSet = Set<Character>()
    var left = 0
    var maxLength = 0
    let chars = Array(s)

    for right in 0..<chars.count {
        while charSet.contains(chars[right]) {
            charSet.remove(chars[left])
            left += 1
        }
        charSet.insert(chars[right])
        maxLength = max(maxLength, right - left + 1)
    }
    return maxLength
}
```

4. Merge Intervals Pattern

Use Cases: Overlapping intervals, scheduling problems

swift

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

    let sorted = intervals.sorted { $0[0] < $1[0] }
    var result = [sorted[0]]

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

        if current[0] <= last[1] {
            last[1] = max(last[1], current[1])
            result[result.count - 1] = last
        } else {
            result.append(current)
        }
    }
    return result
}
```

5. Cyclic Sort Pattern

Use Cases: Problems with numbers in range [1, n]

swift

```swift
func findDisappearedNumbers(_ nums: [Int]) -> [Int] {
    var nums = nums
    var i = 0

    while i < nums.count {
        let correctIndex = nums[i] - 1
        if nums[i] != nums[correctIndex] {
            nums.swapAt(i, correctIndex)
        } else {
            i += 1
        }
    }

    var result = [Int]()
    for i in 0..<nums.count {
        if nums[i] != i + 1 {
            result.append(i + 1)
        }
    }
    return result
}
```

6. In-place Reversal of LinkedList

Use Cases: Reversing linked lists, palindrome checks

swift

```swift
func reverseList(_ head: ListNode?) -> ListNode? {
    var prev: ListNode? = nil
    var current = head

    while current != nil {
        let next = current?.next
        current?.next = prev
        prev = current
        current = next
    }
    return prev
}
```

7. Tree Breadth First Search (BFS)

Use Cases: Level order traversal, minimum depth

swift

```swift
func levelOrder(_ root: TreeNode?) -> [[Int]] {
    guard let root = root else { return [] }

    var result = [[Int]]()
    var queue = [root]

    while !queue.isEmpty {
        let levelSize = queue.count
        var currentLevel = [Int]()

        for _ in 0..<levelSize {
            let node = queue.removeFirst()
            currentLevel.append(node.val)

            if let left = node.left { queue.append(left) }
            if let right = node.right { queue.append(right) }
        }
        result.append(currentLevel)
    }
    return result
}
```

8. Tree Depth First Search (DFS)

Use Cases: Path sum, tree traversals

swift

```swift
func hasPathSum(_ root: TreeNode?, _ targetSum: Int) -> Bool {
    guard let root = root else { return false }

    if root.left == nil && root.right == nil {
        return root.val == targetSum
    }

    let remainingSum = targetSum - root.val
    return hasPathSum(root.left, remainingSum) || hasPathSum(root.right, remainingSum)
}
```

9. Two Heaps Pattern

Use Cases: Find median, smallest/largest elements

swift

```swift
class MedianFinder {
    private var maxHeap = [Int]() // smaller half
    private var minHeap = [Int]() // larger half

    func addNum(_ num: Int) {
        if maxHeap.isEmpty || num <= maxHeap[0] {
            maxHeap.append(num)
            maxHeap.sort(by: >)
        } else {
            minHeap.append(num)
            minHeap.sort()
        }

        // Balance heaps
        if maxHeap.count > minHeap.count + 1 {
            minHeap.append(maxHeap.removeFirst())
            minHeap.sort()
        } else if minHeap.count > maxHeap.count + 1 {
            maxHeap.append(minHeap.removeFirst())
            maxHeap.sort(by: >)
        }
    }

    func findMedian() -> Double {
        if maxHeap.count == minHeap.count {
            return Double(maxHeap[0] + minHeap[0]) / 2.0
        }
        return Double(maxHeap.count > minHeap.count ? maxHeap[0] : minHeap[0])
    }
}
```

10. Top K Elements Pattern

Use Cases: Kth largest/smallest, top K frequent

swift

```swift
func topKFrequent(_ nums: [Int], _ k: Int) -> [Int] {
    var frequency = [Int: Int]()
    for num in nums {
        frequency[num, default: 0] += 1
    }

    return frequency.sorted { $0.value > $1.value }
                .prefix(k)
                .map { $0.key }
}
```

11. K-way Merge Pattern

Use Cases: Merge K sorted lists/arrays

swift

```swift
func mergeKLists(_ lists: [ListNode?]) -> ListNode? {
    guard !lists.isEmpty else { return nil }

    var lists = lists
    while lists.count > 1 {
        var mergedLists = [ListNode?]()

        for i in stride(from: 0, to: lists.count, by: 2) {
            let l1 = lists[i]
            let l2 = i + 1 < lists.count ? lists[i + 1] : nil
            mergedLists.append(mergeTwoLists(l1, l2))
        }
        lists = mergedLists
    }
    return lists[0]
}

func mergeTwoLists(_ l1: ListNode?, _ l2: ListNode?) -> ListNode? {
    let dummy = ListNode(0)
    var current = dummy
    var l1 = l1, l2 = l2

    while l1 != nil && l2 != nil {
        if l1!.val <= l2!.val {
            current.next = l1
            l1 = l1?.next
        } else {
            current.next = l2
            l2 = l2?.next
        }
        current = current.next!
    }

    current.next = l1 ?? l2
    return dummy.next
}
```

12. Dynamic Programming Patterns

Use Cases: Optimization problems, counting problems

Fibonacci Pattern

swift

```swift
func climbStairs(_ n: Int) -> Int {
    if n <= 2 { return n }

    var dp = Array(repeating: 0, count: n + 1)
    dp[1] = 1
    dp[2] = 2

    for i in 3...n {
        dp[i] = dp[i-1] + dp[i-2]
    }
    return dp[n]
}
```

0/1 Knapsack Pattern

swift

```swift
func knapsack(_ weights: [Int], _ values: [Int], _ capacity: Int) -> Int {
    let n = weights.count
    var dp = Array(repeating: Array(repeating: 0, count: capacity + 1), count: n + 1)

    for i in 1...n {
        for w in 1...capacity {
            if weights[i-1] <= w {
                dp[i][w] = max(dp[i-1][w], dp[i-1][w-weights[i-1]] + values[i-1])
            } else {
                dp[i][w] = dp[i-1][w]
            }
        }
    }
    return dp[n][capacity]
}
```

13. Backtracking Pattern

Use Cases: Generating permutations, combinations, solving puzzles

swift

```swift
func permute(_ nums: [Int]) -> [[Int]] {
    var result = [[Int]]()
    var currentPermutation = [Int]()
    var used = Array(repeating: false, count: nums.count)

    func backtrack() {
        if currentPermutation.count == nums.count {
            result.append(currentPermutation)
            return
        }

        for i in 0..<nums.count {
            if used[i] { continue }

            currentPermutation.append(nums[i])
            used[i] = true
            backtrack()
            currentPermutation.removeLast()
            used[i] = false
        }
    }

    backtrack()
    return result
}
```

14. Trie Pattern

Use Cases: Prefix matching, word search

swift

```swift
class TrieNode {
    var children = [Character: TrieNode]()
    var isEndOfWord = false
}

class Trie {
    private let root = TrieNode()

    func insert(_ word: String) {
        var current = root
        for char in word {
            if current.children[char] == nil {
                current.children[char] = TrieNode()
            }
            current = current.children[char]!
        }
        current.isEndOfWord = true
    }

    func search(_ word: String) -> Bool {
        var current = root
        for char in word {
            guard let node = current.children[char] else { return false }
            current = node
        }
        return current.isEndOfWord
    }
}
```

15. Union Find Pattern

Use Cases: Connected components, cycle detection in undirected graphs

swift

```swift
class UnionFind {
    private var parent: [Int]
    private var rank: [Int]

    init(_ n: Int) {
        parent = Array(0..<n)
        rank = Array(repeating: 0, count: n)
    }

    func find(_ x: Int) -> Int {
        if parent[x] != x {
            parent[x] = find(parent[x])
        }
        return parent[x]
    }

    func union(_ x: Int, _ y: Int) -> Bool {
        let rootX = find(x)
        let rootY = find(y)

        if rootX == rootY { return false }

        if rank[rootX] < rank[rootY] {
            parent[rootX] = rootY
        } else if rank[rootX] > rank[rootY] {
            parent[rootY] = rootX
        } else {
            parent[rootY] = rootX
            rank[rootX] += 1
        }
        return true
    }
}
```

System Design Concepts

Low Level Design (LLD) Patterns

1. SOLID Principles

- **Single Responsibility**: Each class should have one reason to change
- **Open/Closed**: Open for extension, closed for modification
- **Liskov Substitution**: Subtypes must be substitutable for base types
- **Interface Segregation**: Many specific interfaces are better than one general interface

- **Dependency Inversion**: Depend on abstractions, not concretions

2. Design Patterns in Swift

Singleton Pattern

swift

```swift
class DatabaseManager {
    static let shared = DatabaseManager()
    private init() {}

    func connect() { /* implementation */ }
}
```

Factory Pattern

swift

```swift
protocol Vehicle {
    func start()
}

class Car: Vehicle {
    func start() { print("Car started") }
}

class Bike: Vehicle {
    func start() { print("Bike started") }
}

class VehicleFactory {
    static func createVehicle(_ type: String) -> Vehicle? {
        switch type {
        case "car": return Car()
        case "bike": return Bike()
        default: return nil
        }
    }
}
```

Observer Pattern

swift

```
protocol Observer {
    func update(_ message: String)
}

class Subject {
    private var observers = [Observer]()

    func addObserver(_ observer: Observer) {
        observers.append(observer)
    }

    func notifyObservers(_ message: String) {
        observers.forEach { $0.update(message) }
    }
}
```

3. Common LLD Problems

- **Parking Lot System**: Multi-level parking with different vehicle types
- **Elevator System**: Multiple elevators with scheduling algorithms
- **Chat System**: Real-time messaging with user management
- **Library Management**: Book checkout/return with user accounts
- **ATM System**: Cash withdrawal with account management
- **Restaurant Management**: Order processing with kitchen workflow

High Level Design (HLD) Concepts

1. System Architecture Components

- **Load Balancers**: Distribute incoming requests across multiple servers
- **CDN**: Content Delivery Network for static asset distribution
- **Caching**: Redis/Memcached for frequently accessed data
- **Message Queues**: RabbitMQ/Kafka for asynchronous processing
- **Databases**: SQL (MySQL/PostgreSQL) vs NoSQL (MongoDB/Cassandra)
- **Microservices**: Service decomposition and communication

2. Scalability Patterns

- **Horizontal Scaling**: Add more servers
- **Vertical Scaling**: Increase server capacity
- **Database Sharding**: Distribute data across multiple databases

- **Replication**: Master-slave database setup
- **Federation**: Split databases by function

3. Common HLD Problems

- **URL Shortener (bit.ly)**: Short URL generation and redirection
- **Social Media Feed**: Timeline generation and content distribution
- **Chat Application**: Real-time messaging at scale
- **Video Streaming**: Content delivery and encoding
- **Ride Sharing**: Location-based matching and routing
- **E-commerce**: Product catalog and order processing
- **Search Engine**: Web crawling and indexing

4. System Design Interview Framework

1. **Clarify Requirements**: Functional and non-functional requirements
2. **Estimate Scale**: Users, requests per second, storage needs
3. **Design High-Level Architecture**: Major components and data flow
4. **Database Design**: Schema design and data modeling
5. **API Design**: REST endpoints and request/response formats
6. **Detailed Design**: Deep dive into critical components
7. **Scale the Design**: Handle increased load and traffic
8. **Address Bottlenecks**: Identify and resolve performance issues

Swift-Specific Interview Topics

1. Memory Management

- **ARC (Automatic Reference Counting)**: How Swift manages memory
- **Strong, Weak, Unowned References**: Preventing retain cycles
- **Memory Leaks**: Common causes and prevention

2. Concurrency

- **Grand Central Dispatch (GCD)**: Queue management and threading
- **async/await**: Modern asynchronous programming
- **Actors**: Thread-safe state management
- **Combine Framework**: Reactive programming

3. Swift Language Features

- **Optionals**: Safe handling of nil values
- **Generics**: Type-safe flexible code
- **Protocols**: Interface-oriented programming
- **Extensions**: Adding functionality to existing types
- **Error Handling**: try/catch and Result types

4. iOS Development Concepts

- **MVC/MVVM/VIPER**: Architectural patterns
- **Delegation**: Communication between objects
- **Notifications**: Broadcast communication
- **Core Data**: Object-relational mapping
- **Networking**: URLSession and API integration

Interview Preparation Strategy

1. Practice Schedule

- **Week 1-2**: Master two pointers, sliding window, and arrays
- **Week 3-4**: Trees, graphs, and BFS/DFS
- **Week 5-6**: Dynamic programming and backtracking
- **Week 7-8**: System design fundamentals and common patterns
- **Week 9-10**: Mock interviews and weak area improvement

2. Problem-Solving Approach

1. **Understand the Problem**: Read carefully, ask clarifying questions
2. **Think of Examples**: Walk through test cases
3. **Design Algorithm**: Start with brute force, then optimize
4. **Code Implementation**: Write clean, readable code
5. **Test and Debug**: Check edge cases and boundary conditions
6. **Optimize**: Improve time and space complexity

3. System Design Interview Tips

- **Start with Requirements**: Always clarify what you're building
- **Think Big Picture**: Design for scale from the beginning
- **Be Specific**: Provide concrete numbers and technologies
- **Consider Trade-offs**: Discuss different approaches and their pros/cons

- **Stay Organized**: Use diagrams and structured thinking

4. Swift Interview Tips

- **Showcase Language Knowledge**: Demonstrate Swift-specific features
- **Memory Management**: Always consider ARC and potential retain cycles
- **Error Handling**: Use proper Swift error handling patterns
- **Protocol-Oriented Programming**: Show understanding of Swift's philosophy
- **Performance**: Discuss optimization techniques and best practices

Remember to practice coding problems daily, understand the underlying concepts rather than memorizing solutions, and always consider the trade-offs in your design decisions. Good luck with your interviews!