# Comprehensive Swift Programming Guide

***Swift Language • SwiftUI • Combine • Networking***
***iOS Development • Data Structures & Algorithms***

A complete guide with 120+ topics, code examples, and practical implementations

Generated on: September 13, 2025

# Table of Contents

**PART I: SWIFT LANGUAGE FUNDAMENTALS**

Chapter 1: Swift Basics

1.1 Variables and Constants

1.2 Data Types

1.3 Optionals

1.4 Control Flow

1.5 Functions

1.6 Closures

1.7 Collections

1.8 Strings

1.9 Error Handling

**Chapter 2: Object-Oriented Programming**

2.1 Classes vs Structures

2.2 Properties

2.3 Inheritance

2.4 Initialization

2.5 Protocols

2.6 Extensions

Chapter 3: Advanced Swift

**3.1 Generics**

3.2 Memory Management

3.3 Protocol-Oriented Programming

3.4 Property Wrappers

3.5 Result Builders

PART II: CONCURRENCY & MODERN SWIFT

**Chapter 4: Concurrency**

4.1 Async/Await

4.2 Tasks

4.3 Actors

4.4 Structured Concurrency

PART III: SwiftUI FRAMEWORK

Chapter 5: SwiftUI Fundamentals

**5.1 Views and Modifiers**

5.2 Layout System

5.3 State Management

5.4 Navigation

5.5 Lists and Forms

Chapter 6: Advanced SwiftUI

6.1 Custom Views

6.2 Animations

6.3 Gestures

**6.4 MVVM Pattern**

PART IV: REACTIVE PROGRAMMING

Chapter 7: Combine Framework

7.1 Publishers and Subscribers

7.2 Operators

7.3 Subjects

7.4 Schedulers

PART V: NETWORKING & APIs

**Chapter 8: Networking**

8.1 URLSession

8.2 JSON & Codable

8.3 REST APIs

8.4 Authentication

8.5 WebSockets

PART VI: iOS DEVELOPMENT

Chapter 9: Data Persistence

9.1 UserDefaults

9.2 Core Data

9.3 File System

9.4 Keychain

Chapter 10: Testing & Architecture
10.1 Unit Testing
10.2 UI Testing
10.3 MVVM Architecture
10.4 Dependency Injection

APPENDIX: DATA STRUCTURES & ALGORITHMS
A.1 Array Problems
A.2 Linked List Problems
A.3 Tree Problems
A.4 Graph Problems

# PART I: SWIFT LANGUAGE FUNDAMENTALS

# Chapter 1: Swift Basics

## 1.1 Variables and Constants

Swift uses 'var' for mutable variables and 'let' for immutable constants. Type inference allows Swift to automatically determine types.

### Code Example:

```
// Variables (mutable)
var playerName = "Alice"
var score = 100
var isActive = true

// Constants (immutable)
let maxPlayers = 4
let gameTitle = "Swift Adventure"
let pi = 3.14159

// Explicit type annotations
var temperature: Double = 98.6
let items: [String] = ["sword", "shield", "potion"]

// Multiple declarations
var x = 0.0, y = 0.0, z = 0.0
let red, green, blue: Double
```

### Key Points:

**• Use 'let' by default, 'var' only when you need to change the value**
**• Type inference reduces verbosity while maintaining type safety**
**• Constants improve performance and prevent accidental mutations**

### Notes:

*Swift encourages immutability through 'let'. The compiler optimizes constants more effectively than variables.*

## 1.2 Data Types

Swift provides various built-in data types including integers, floating-point numbers, booleans, strings, and more.

### Code Example:

```
// Integer types
let smallNumber: Int8 = 127
let regularNumber: Int = 42
let bigNumber: Int64 = 9223372036854775807

// Floating-point types
let pi: Float = 3.14159
let precisePi: Double = 3.141592653589793

// Boolean
let isSwiftFun: Bool = true

// Character and String
let letter: Character = "A"
let greeting: String = "Hello, Swift!"

// Type conversion
let integerValue = 42
let floatValue = Float(integerValue)
let stringValue = String(integerValue)

// Type checking
if floatValue is Float {
    print("It's a Float!")
}
```

### Key Points:

**• Int and Double are the most commonly used numeric types**
**• Swift doesn't perform implicit type conversions**
**• Use type conversion initializers for explicit conversions**
**• Type checking with 'is' operator helps ensure type safety**

### Notes:

*Swift is a type-safe language, preventing type-related errors at compile time.*

## 1.3 Optionals

Optionals represent either a value or nil (absence of value). They're fundamental to Swift's safety model.

### Code Example:

```
// Declaring optionals
var optionalString: String? = "Hello"
var optionalInt: Int? = nil

// Optional binding with if-let
if let actualString = optionalString {
    print("The string is: \(actualString)")
} else {
    print("No string value")
}
```

```swift
// Guard statement
func processString(_ str: String?) {
    guard let unwrapped = str else {
        print("String is nil")
        return
    }
    print("Processing: \(unwrapped)")
}

// Nil-coalescing operator
let defaultName = "Anonymous"
let userName = optionalString ?? defaultName

// Optional chaining
class Person {
    var residence: Residence?
}
class Residence {
    var address: String?
}

let person = Person()
let address = person.residence?.address

// Implicitly unwrapped optionals
var assumedString: String! = "An implicitly unwrapped optional string."
```

### Key Points:

**• Use optionals to handle absence of values safely**
**• Prefer optional binding over force unwrapping**
**• Guard statements provide early exit for nil values**
**• Optional chaining prevents crashes when accessing nested optionals**

### Notes:

*Optionals eliminate null pointer exceptions and make your code more robust.*

## 1.4 Control Flow

Swift provides various control flow statements including if, switch, loops, and control transfer statements.

### Code Example:

```swift
// If statements
let temperature = 75
if temperature > 80 {
    print("It's hot!")
} else if temperature > 60 {
    print("It's warm")
} else {
    print("It's cool")
}

// Switch statements (powerful in Swift)
let character = "a"
switch character {
case "a", "e", "i", "o", "u":
    print("It's a vowel")
case "b"..."z":
    print("It's a consonant")
default:
    print("Not a letter")
}

// Switch with ranges and where clauses
let point = (2, 3)
switch point {
case (0, 0):
    print("Origin")
case (_, 0):
    print("On x-axis")
case (0, _):
    print("On y-axis")
case let (x, y) where x == y:
    print("On diagonal")
case let (x, y):
    print("Point at (\(x), \(y))")
}

// For loops
for i in 1...5 {
    print("Count: \(i)")
}

let names = ["Anna", "Alex", "Brian", "Jack"]
for name in names {
    print("Hello, \(name)!")
}

// While loops
var counter = 0
while counter < 3 {
    print(counter)
    counter += 1
}

// Repeat-while (do-while equivalent)
repeat {
    print("This executes at least once")
    counter -= 1
} while counter > 0
```

## Key Points:

**• Swift's switch statement is exhaustive and doesn't fall through by default**
**• Pattern matching in switch makes complex conditions elegant**
**• Range operators (...) and (..<) are useful in loops and switches**
**• Control transfer statements: continue, break, fallthrough, return, throw**

## Notes:

*Swift's control flow statements are more powerful than many other languages, especially switch statements.*

## 1.5 Functions

Functions are self-contained chunks of code that perform specific tasks. Swift functions are flexible and powerful.

## Code Example:

```
// Basic function
func greet(person: String) -> String {
    return "Hello, \(person)!"
}
let greeting = greet(person: "Taylor")

// Function with multiple parameters
func greet(person: String, from hometown: String) -> String {
    return "Hello \(person)! Glad you could visit from \(hometown)."
}
print(greet(person: "Bill", from: "Cupertino"))

// Function with default parameters
func greet(person: String, from hometown: String = "Unknown") -> String {
    return "Hello \(person)! Glad you could visit from \(hometown)."
}

// Variadic parameters
func arithmeticMean(_ numbers: Double...) -> Double {
    var total: Double = 0
    for number in numbers {
        total += number
    }
    return total / Double(numbers.count)
}
print(arithmeticMean(1, 2, 3, 4, 5))

// In-out parameters
func swapTwoInts(_ a: inout Int, _ b: inout Int) {
    let temporaryA = a
    a = b
    b = temporaryA
}

var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt, &anotherInt)

// Function types
func addTwoInts(_ a: Int, _ b: Int) -> Int {
    return a + b
}

let mathFunction: (Int, Int) -> Int = addTwoInts
print(mathFunction(2, 3))

// Nested functions
func chooseStepFunction(backward: Bool) -> (Int) -> Int {
    func stepForward(input: Int) -> Int { return input + 1 }
    func stepBackward(input: Int) -> Int { return input - 1 }

    return backward ? stepBackward : stepForward
}
```

### Key Points:

- **Parameter labels improve code readability**
- **Default parameters reduce function overloading**
- **inout parameters allow functions to modify external variables**
- **Functions are first-class types in Swift**

### Notes:

*Swift functions support many advanced features like closures, higher-order functions, and functional programming patterns.*

## 1.6 Closures

Closures are self-contained blocks of functionality that can be passed around. They're similar to lambdas in other languages.

### Code Example:

```
// Basic closure syntax
let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]

// Full closure syntax
let reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in
    return s1 > s2
})

// Inferring type from context
let reversed1 = names.sorted(by: { s1, s2 in return s1 > s2 })

// Implicit returns
let reversed2 = names.sorted(by: { s1, s2 in s1 > s2 })

// Shorthand argument names
let reversed3 = names.sorted(by: { $0 > $1 })

// Operator method
let reversed4 = names.sorted(by: >)
```

```
// Trailing closure syntax
let reversed5 = names.sorted { $0 > $1 }

// Capturing values
func makeIncrementer(forIncrement amount: Int) -> () -> Int {
    var runningTotal = 0
    func incrementer() -> Int {
        runningTotal += amount
        return runningTotal
    }
    return incrementer
}

let incrementByTen = makeIncrementer(forIncrement: 10)
print(incrementByTen()) // 10
print(incrementByTen()) // 20

// Escaping closures
var completionHandlers: [() -> Void] = []

func someFunctionWithEscapingClosure(completionHandler: @escaping () -> Void) {
    completionHandlers.append(completionHandler)
}

// Autoclosures
func simpleAssert(_ condition: @autoclosure () -> Bool, _ message: String) {
    if !condition() {
        print(message)
    }
}

let testNumber = 5
simpleAssert(testNumber > 0, "Number must be positive")
```

### Key Points:

**• Closures can capture and store references to variables and constants**
**• Trailing closure syntax makes code more readable**
**• @escaping closures outlive the function that calls them**
**• @autoclosure automatically wraps expressions in closures**

### Notes:

*Closures are extensively used in Swift for callbacks, functional programming, and asynchronous operations.*

## 1.7 Collections

Swift provides three primary collection types: arrays, sets, and dictionaries for storing multiple values.

### Code Example:

```
// Arrays
var fruits = ["apple", "banana", "orange"]
fruits.append("grape")
fruits.insert("kiwi", at: 1)

// Array methods
let numbers = [1, 2, 3, 4, 5]
let doubled = numbers.map { $0 * 2 }
let evens = numbers.filter { $0 % 2 == 0 }
let sum = numbers.reduce(0, +)

// Sets
var uniqueNumbers: Set<Int> = [1, 2, 3, 2, 1]
print(uniqueNumbers) // [1, 2, 3]

let set1: Set = [1, 2, 3]
let set2: Set = [3, 4, 5]
let intersection = set1.intersection(set2) // [3]
let union = set1.union(set2) // [1, 2, 3, 4, 5]

// Dictionaries
var studentGrades = ["Alice": 95, "Bob": 87, "Charlie": 92]
studentGrades["Diana"] = 89
studentGrades.updateValue(88, forKey: "Bob")

// Dictionary iteration
for (name, grade) in studentGrades {
    print("\(name): \(grade)")
}

// Nested collections
let matrix: [[Int]] = [[1, 2], [3, 4], [5, 6]]
let coordinates = [(x: 1, y: 2), (x: 3, y: 4)]
```

### Key Points:

**• Arrays are ordered collections of values**
**• Sets store unique values in no defined ordering**
**• Dictionaries store key-value associations**
**• All collections support functional programming methods**

### Notes:

*Swift collections are type-safe and provide powerful methods for data manipulation.*

## 1.8 Strings

Swift strings are Unicode-compliant and provide powerful manipulation methods.

### Code Example:

```
// String basics
let greeting = "Hello, World!"
let multilineString = """
    This is a multiline
    string in Swift with
    proper formatting
    """

// String interpolation
let name = "Swift"
let version = 5.0
let message = "Welcome to \(name) \(version)!"

// String methods
let text = "Hello, Swift Programming"
print(text.count) // Character count
print(text.uppercased())
print(text.lowercased())
print(text.hasPrefix("Hello"))
print(text.hasSuffix("Programming"))

// String manipulation
let sentence = "Swift is awesome"
let words = sentence.split(separator: " ")
let joined = words.joined(separator: "-")

// Character iteration
for character in greeting {
    print(character)
}

// String indices
let str = "Swift"
let startIndex = str.startIndex
let endIndex = str.endIndex
let secondChar = str[str.index(after: startIndex)]

// String slicing
let range = str.index(str.startIndex, offsetBy: 1)..<str.index(str.endIndex, offsetBy: -1)
let substring = str[range]

// Regular expressions (iOS 16+)
let pattern = #"\d+"#
if let regex = try? Regex(pattern) {
    let numbers = "Age: 25, Score: 100"
    let matches = numbers.matches(of: regex)
}
```

## Key Points:

**• Strings are value types and use copy-on-write optimization**
**• String interpolation with \() is preferred over concatenation**
**• Strings use String.Index for position-based operations**
**• Regular expressions provide powerful pattern matching**

## Notes:

*Swift strings are designed for Unicode correctness and international text support.*

## 1.9 Error Handling

Swift provides first-class error handling with do-catch blocks, throwing functions, and the Result type.

### Code Example:

```
// Define errors
enum ValidationError: Error {
    case tooShort
    case tooLong
    case invalidCharacters
    case empty
}

// Throwing function
func validatePassword(_ password: String) throws -> Bool {
    if password.isEmpty {
        throw ValidationError.empty
    }

    if password.count < 8 {
        throw ValidationError.tooShort
    }

    if password.count > 50 {
        throw ValidationError.tooLong
    }

    return true
}

// Do-catch block
func testPassword() {
    do {
        try validatePassword("secret")
        print("Password is valid")
    } catch ValidationError.tooShort {
        print("Password is too short")
    } catch ValidationError.empty {
        print("Password cannot be empty")
    } catch {
        print("Unknown error: \(error)")
    }
}

// Try variants
let password1 = try? validatePassword("mypassword") // Returns nil on error
let password2 = try! validatePassword("validpassword") // Crashes on error

// Result type
func validatePasswordResult(_ password: String) -> Result<Bool, ValidationError> {
    do {
        let isValid = try validatePassword(password)
        return .success(isValid)
    } catch let error as ValidationError {
        return .failure(error)
    } catch {
        return .failure(.invalidCharacters)
    }
}

// Using Result
let result = validatePasswordResult("test")
switch result {
case .success(let isValid):
    print("Validation result: \(isValid)")
case .failure(let error):
    print("Validation failed: \(error)")
}

// Rethrowing functions
func processPasswords<T>(_ passwords: [String], processor: (String) throws -> T) rethrows ->
    return try passwords.map(processor)
}
```

## Key Points:

**• Use specific error types conforming to Error protocol**
**• Do-catch blocks handle errors gracefully**
**• try? converts errors to optionals, try! force-unwraps**
**• Result type provides functional error handling**

## Notes:

*Swift's error handling is designed to be explicit and safe, preventing runtime crashes from unhandled errors.*

# Chapter 2: Object-Oriented Programming

## 2.1 Classes vs Structures

Swift provides both classes and structures. Understanding when to use each is crucial for effective Swift programming.

### Code Example:

```
// Structure (Value Type)
struct Point {
    var x: Double
    var y: Double

    func distanceFromOrigin() -> Double {
        return sqrt(x * x + y * y)
    }

    // Mutating method for value types
    mutating func moveBy(x deltaX: Double, y deltaY: Double) {
        x += deltaX
        y += deltaY
    }
}

// Class (Reference Type)
class Vehicle {
    var currentSpeed = 0.0
    var description: String {
        return "traveling at \(currentSpeed) miles per hour"
    }

    func makeNoise() {
        // Override in subclass
    }
}

class Bicycle: Vehicle {
    var hasBasket = false

    override func makeNoise() {
        print("Ring ring!")
    }
}

// Identity operators for reference types
let vehicle1 = Vehicle()
let vehicle2 = Vehicle()
let vehicle3 = vehicle1

if vehicle1 === vehicle3 {
    print("Same instance")
}

// Copy behavior difference
var point1 = Point(x: 1.0, y: 2.0)
var point2 = point1  // Copies the value
point2.x = 3.0
print(point1.x)  // Still 1.0

let bike1 = Bicycle()
let bike2 = bike1  // Same reference
bike2.currentSpeed = 10.0
print(bike1.currentSpeed)  // Also 10.0
```

### Key Points:

**• Structures are value types (copied), classes are reference types (shared)**
**• Use structures for simple data containers and value semantics**
**• Use classes when you need inheritance or reference semantics**
**• Identity operators (=== and !==) compare reference equality**

### Notes:

*Choose structures by default and classes when you specifically need reference semantics.*

## 2.2 Properties

Properties associate values with classes, structures, and enumerations. Swift provides stored and computed properties.

### Code Example:

```
// Stored properties
struct FixedLengthRange {
    var firstValue: Int
    let length: Int  // Constant stored property
}

// Lazy stored properties
class DataImporter {
    var filename = "data.txt"
    // Expensive initialization
}

class DataManager {
    lazy var importer = DataImporter()
    var data: [String] = []
}

// Computed properties
struct Point {
    var x = 0.0, y = 0.0
```

```swift
}

struct Size {
    var width = 0.0, height = 0.0
}

struct Rect {
    var origin = Point()
    var size = Size()

    var center: Point {
        get {
            let centerX = origin.x + (size.width / 2)
            let centerY = origin.y + (size.height / 2)
            return Point(x: centerX, y: centerY)
        }
        set(newCenter) {
            origin.x = newCenter.x - (size.width / 2)
            origin.y = newCenter.y - (size.height / 2)
        }
    }

    // Read-only computed property
    var area: Double {
        return size.width * size.height
    }
}

// Property observers
class StepCounter {
    var totalSteps: Int = 0 {
        willSet(newTotalSteps) {
            print("About to set totalSteps to \(newTotalSteps)")
        }
        didSet {
            if totalSteps > oldValue {
                print("Added \(totalSteps - oldValue) steps")
            }
        }
    }
}

// Property wrappers
@propertyWrapper
struct Clamped<T: Comparable> {
    private var value: T
    private let range: ClosedRange<T>

    init(wrappedValue: T, _ range: ClosedRange<T>) {
        self.range = range
        self.value = max(range.lowerBound, min(range.upperBound, wrappedValue))
    }

    var wrappedValue: T {
        get { value }
        set { value = max(range.lowerBound, min(range.upperBound, newValue)) }
    }
```

```
}

struct Player {
    @Clamped(0...100) var health: Int = 100
    @Clamped(0...10) var level: Int = 1
}
```

### Key Points:

- **Stored properties store constant and variable values**
- **Computed properties calculate values on-the-fly**
- **Property observers respond to changes in property values**
- **Property wrappers provide reusable property behavior**

### Notes:

*Properties are a fundamental part of Swift's type system, providing flexible data access patterns.*

## 2.3 Inheritance

Classes can inherit methods, properties, and characteristics from another class. Swift supports single inheritance.

### Code Example:

```
// Base class
class Vehicle {
    var currentSpeed = 0.0

    var description: String {
        return "traveling at \(currentSpeed) miles per hour"
    }

    func makeNoise() {
        // Default implementation
        print("Some generic vehicle noise")
    }
}

// Subclass
class Bicycle: Vehicle {
    var hasBasket = false

    override func makeNoise() {
        print("Ring ring!")
    }
}

// Further subclassing
class Tandem: Bicycle {
    var currentNumberOfPassengers = 0
    override var description: String {
        return super.description + " with \(currentNumberOfPassengers) passengers"
    }
}

// Preventing inheritance
final class FinalVehicle: Vehicle {
    // Cannot be subclassed
}

// Overriding properties
class Car: Vehicle {
    var gear = 1

    override var description: String {
        return super.description + " in gear \(gear)"
    }

    // Overriding property observers
    override var currentSpeed: Double {
        didSet {
            gear = Int(currentSpeed / 10.0) + 1
        }
    }
}

// Initialization inheritance
class ElectricCar: Car {
    var batteryLevel: Double

    init(batteryLevel: Double) {
        self.batteryLevel = batteryLevel
        super.init()
        self.currentSpeed = 25.0
    }

    override func makeNoise() {
        print("Whisper quiet...")
    }
}
```

## Key Points:

- **Only classes support inheritance in Swift**
- **Use 'override' keyword to override methods and properties**
- **Call superclass methods with 'super'**
- **Use 'final' to prevent inheritance**

## Notes:

*Inheritance enables code reuse and polymorphism, but favor composition over inheritance when possible.*

## 2.4 Initialization

Initialization is the process of preparing an instance for use. Swift provides designated and convenience initializers.

### Code Example:

```
// Basic initialization
struct Celsius {
    var temperatureInCelsius: Double

    init(fromFahrenheit fahrenheit: Double) {
        temperatureInCelsius = (fahrenheit - 32.0) / 1.8
    }

    init(fromKelvin kelvin: Double) {
        temperatureInCelsius = kelvin - 273.15
    }

    init(_ celsius: Double) {
        temperatureInCelsius = celsius
    }
}

// Class initialization
class Food {
    var name: String

    init(name: String) {
        self.name = name
    }

    convenience init() {
        self.init(name: "[Unnamed]")
    }
}

class RecipeIngredient: Food {
    var quantity: Int

    init(name: String, quantity: Int) {
        self.quantity = quantity
        super.init(name: name)
    }

    override convenience init(name: String) {
        self.init(name: name, quantity: 1)
    }
}

// Failable initializers
struct Animal {
    let species: String

    init?(species: String) {
        if species.isEmpty {
            return nil
        }
        self.species = species
    }
}

// Required initializers
class SomeClass {
    required init() {
        // Implementation
    }
}

class SomeSubclass: SomeClass {
    required init() {
        // Must implement required initializer
    }
}

// Memberwise initializers (structs only)
struct Point {
    var x: Double
    var y: Double
    // Automatically gets init(x:y:)
}

// Deinitialization
class Player {
    let playerName: String

    init(name: String) {
        self.playerName = name
        print("\(playerName) has joined the game")
    }

    deinit {
        print("\(playerName) has left the game")
    }
}
```

## Key Points:

**• Designated initializers fully initialize all properties**
**• Convenience initializers call other initializers**
**• Failable initializers return nil if initialization fails**
**• Required initializers must be implemented by all subclasses**

## Notes:

*Swift's initialization system ensures all properties are initialized before the instance is ready for use.*

# Chapter 3: Advanced Swift

## 3.1 Generics

Generics enable you to write flexible, reusable functions and types that can work with any type, subject to requirements you define.

### Code Example:

```
// Generic function
func swapTwoValues<T>(_ a: inout T, _ b: inout T) {
    let temporaryA = a
    a = b
    b = temporaryA
}

var someInt = 3
var anotherInt = 107
swapTwoValues(&someInt, &anotherInt)

// Generic types
struct Stack<Element> {
    var items: [Element] = []

    mutating func push(_ item: Element) {
        items.append(item)
    }

    mutating func pop() -> Element {
        return items.removeLast()
    }

    func peek() -> Element? {
        return items.last
    }
}

var stackOfStrings = Stack<String>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")

// Type constraints
func findIndex<T: Equatable>(of valueToFind: T, in array: [T]) -> Int? {
    for (index, value) in array.enumerated() {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

```
// Associated types in protocols
protocol Container {
    associatedtype Item
    mutating func append(_ item: Item)
    var count: Int { get }
    subscript(i: Int) -> Item { get }
}

struct IntStack: Container {
    typealias Item = Int
    var items: [Int] = []

    mutating func append(_ item: Int) {
        items.append(item)
    }

    var count: Int {
        return items.count
    }

    subscript(i: Int) -> Int {
        return items[i]
    }
}

// Generic where clauses
func allItemsMatch<C1: Container, C2: Container>
    (_ someContainer: C1, _ anotherContainer: C2) -> Bool
    where C1.Item == C2.Item, C1.Item: Equatable {

    if someContainer.count != anotherContainer.count {
        return false
    }

    for i in 0..<someContainer.count {
        if someContainer[i] != anotherContainer[i] {
            return false
        }
    }

    return true
}
```

## Key Points:

- **Generics provide type safety while maintaining flexibility**
- **Type constraints ensure generic types conform to required protocols**
- **Associated types make protocols more flexible**
- **Where clauses add additional requirements to generic functions**

## Notes:

*Generics are extensively used in Swift's standard library and are key to creating reusable, type-safe code.*

## 3.2 Protocols

Protocols define a blueprint of methods, properties, and requirements that suit a particular task or piece of functionality.

### Code Example:

```
// Basic protocol
protocol Drawable {
    func draw()
    var area: Double { get }
    var perimeter: Double { get }
}

// Protocol implementation
struct Circle: Drawable {
    let radius: Double

    func draw() {
        print("Drawing a circle with radius \(radius)")
    }

    var area: Double {
        return .pi * radius * radius
    }

    var perimeter: Double {
        return 2 * .pi * radius
    }
}

// Protocol inheritance
protocol Shape3D: Drawable {
    var volume: Double { get }
}

// Multiple protocol conformance
protocol Identifiable {
    var id: String { get }
}

struct User: Identifiable, CustomStringConvertible {
    let id: String
    let name: String

    var description: String {
        return "User(id: \(id), name: \(name))"
    }
}

// Protocol extensions
extension Drawable {
    func drawWithBorder() {
        print("Drawing border...")
        draw()
        print("Border complete")
    }

    // Default implementation
    var description: String {
        return "A shape with area \(area)"
    }
}

// Protocol with associated types
protocol Container {
    associatedtype Item
    var count: Int { get }
    mutating func append(_ item: Item)
    subscript(i: Int) -> Item { get }
}

struct Stack<Element>: Container {
    var items: [Element] = []

    var count: Int {
        return items.count
    }

    mutating func append(_ item: Element) {
        items.append(item)
    }

    subscript(i: Int) -> Element {
        return items[i]
    }
}

// Protocol composition
protocol Named {
    var name: String { get }
}

protocol Aged {
    var age: Int { get }
}

func greetPerson(_ person: Named & Aged) {
    print("Hello, \(person.name), you are \(person.age) years old")
}

// Checking protocol conformance
if let circle = someObject as? Drawable {
    circle.draw()
}
```

## Key Points:

**• Protocols define contracts that types must fulfill**
**• Protocol extensions provide default implementations**
**• Associated types make protocols generic**
**• Protocol composition combines multiple protocols**

### Notes:

*Protocols are fundamental to Swift's protocol-oriented programming paradigm.*

## 3.3 Extensions

Extensions add new functionality to existing classes, structures, enumerations, or protocol types without modifying their source code.

### Code Example:

```
// Basic extension
extension Double {
    var squared: Double {
        return self * self
    }

    func rounded(toDecimalPlaces places: Int) -> Double {
        let multiplier = pow(10, Double(places))
        return (self * multiplier).rounded() / multiplier
    }
}

let number = 3.14159
print(number.squared) // 9.8696
print(number.rounded(toDecimalPlaces: 2)) // 3.14

// Extension with initializers
extension String {
    init(repeating character: Character, count: Int) {
        self = String(Array(repeating: character, count: count))
    }

    var isPalindrome: Bool {
        let cleaned = self.lowercased().filter { $0.isLetter }
        return cleaned == String(cleaned.reversed())
    }
}

let stars = String(repeating: "*", count: 5) // "*****"
print("racecar".isPalindrome) // true

// Extension with subscripts
extension Array {
    subscript(safe index: Int) -> Element? {
        return indices.contains(index) ? self[index] : nil
    }
}

let numbers = [1, 2, 3, 4, 5]
print(numbers[safe: 10]) // nil instead of crash

// Extension with nested types
```

```
extension Character {
    enum Kind {
        case vowel
        case consonant
        case other
    }

    var kind: Kind {
        switch lowercased() {
        case "a", "e", "i", "o", "u":
            return .vowel
        case "a"..."z":
            return .consonant
        default:
            return .other
        }
    }
}

let char: Character = "E"
print(char.kind) // vowel

// Generic extension
extension Array where Element: Comparable {
    func quickSorted() -> [Element] {
        guard count > 1 else { return self }

        let pivot = self[count / 2]
        let less = self.filter { $0 < pivot }
        let equal = self.filter { $0 == pivot }
        let greater = self.filter { $0 > pivot }

        return less.quickSorted() + equal + greater.quickSorted()
    }
}

let unsorted = [3, 1, 4, 1, 5, 9, 2, 6]
let sorted = unsorted.quickSorted()
```

## Key Points:

- **Extensions add functionality without modifying original code**
- **Can add computed properties, methods, initializers, and subscripts**
- **Generic extensions can add conditional functionality**
- **Extensions can conform types to protocols**

## Notes:

*Extensions are a powerful way to organize code and add functionality to existing types.*

# PART II: CONCURRENCY & MODERN SWIFT

# Chapter 4: Concurrency

## 4.1 Async/Await

Swift's async/await syntax provides a clean way to write asynchronous code that reads like synchronous code.

### Code Example:

```
// Basic async function
func fetchUserData(id: String) async throws -> User {
    let url = URL(string: "https://api.example.com/users/\(id)")!
    let (data, _) = try await URLSession.shared.data(from: url)
    return try JSONDecoder().decode(User.self, from: data)
}

// Calling async functions
func loadUserProfile() async {
    do {
        let user = try await fetchUserData(id: "123")
        print("Loaded user: \(user.name)")
    } catch {
        print("Failed to load user: \(error)")
    }
}

// Async properties
class ImageLoader {
    private var cache: [URL: UIImage] = [:]

    var imageCount: Int {
        cache.count
    }

    func image(from url: URL) async throws -> UIImage {
        // Check cache first
        if let cachedImage = cache[url] {
            return cachedImage
        }

        // Download image
        let (data, _) = try await URLSession.shared.data(from: url)
        guard let image = UIImage(data: data) else {
            throw ImageError.invalidData
        }

        // Cache the image
        cache[url] = image
        return image
    }
}
```

```swift
// Async sequences
func countDown(from number: Int) -> AsyncStream<Int> {
    AsyncStream { continuation in
        Task {
            for i in (0...number).reversed() {
                continuation.yield(i)
                try await Task.sleep(nanoseconds: 1_000_000_000) // 1 second
            }
            continuation.finish()
        }
    }
}

// Using async sequences
func runCountdown() async {
    for await count in countDown(from: 5) {
        print("Count: \(count)")
    }
    print("Done!")
}

// Async/await with completion handlers
func legacyNetworkCall(completion: @escaping (Result<Data, Error>) -> Void) {
    // Legacy callback-based code
}

// Convert to async/await
func modernNetworkCall() async throws -> Data {
    return try await withCheckedThrowingContinuation { continuation in
        legacyNetworkCall { result in
            continuation.resume(with: result)
        }
    }
}

// Multiple concurrent operations
func loadMultipleUsers() async throws -> [User] {
    async let user1 = fetchUserData(id: "1")
    async let user2 = fetchUserData(id: "2")
    async let user3 = fetchUserData(id: "3")

    return try await [user1, user2, user3]
}
```

## Key Points:

**• async functions must be called with await**
**• async/await eliminates callback hell**
**• Use async let for concurrent operations**
**• AsyncSequence provides asynchronous iteration**

## Notes:

*Async/await makes asynchronous code more readable and easier to debug than callback-based approaches.*

## 4.2 Tasks

Tasks represent units of asynchronous work. Swift provides Task, TaskGroup, and various cancellation mechanisms.

### Code Example:

```
// Basic Task creation
func startBackgroundWork() {
    Task {
        let result = await performLongRunningOperation()
        await updateUI(with: result)
    }
}

// Task with priority
func highPriorityWork() {
    Task(priority: .high) {
        await performCriticalOperation()
    }
}

// Detached tasks
func detachedWork() {
    Task.detached {
        // This task doesn't inherit context
        await performIndependentWork()
    }
}

// TaskGroup for multiple concurrent operations
func processItemsConcurrently(_ items: [String]) async -> [ProcessedItem] {
    await withTaskGroup(of: ProcessedItem.self) { group in
        var results: [ProcessedItem] = []

        for item in items {
            group.addTask {
                return await processItem(item)
            }
        }

        for await result in group {
            results.append(result)
        }

        return results
    }
}

// Error handling in TaskGroup
func processWithErrorHandling(_ items: [String]) async -> [ProcessedItem] {
    await withTaskGroup(of: Result<ProcessedItem, Error>.self) { group in
        var results: [ProcessedItem] = []
        for item in items {
            group.addTask {
                do {
                    let processed = try await processItemThrowing(item)
                    return .success(processed)
                } catch {
                    return .failure(error)
                }
            }
        }

        for await result in group {
            switch result {
            case .success(let item):
                results.append(item)
            case .failure(let error):
                print("Failed to process item: \(error)")
            }
        }

        return results
    }
}

// Task cancellation
class DataProcessor {
    private var currentTask: Task<Void, Error>?

    func startProcessing() {
        currentTask = Task {
            for i in 1...1000 {
                // Check for cancellation
                try Task.checkCancellation()

                await processItem(i)

                // Alternative cancellation check
                if Task.isCancelled {
                    print("Task was cancelled")
                    return
                }
            }
        }
    }

    func cancelProcessing() {
        currentTask?.cancel()
    }
}

// Task local values
enum TaskLocals {
    @TaskLocal static var userID: String?
    @TaskLocal static var requestID: String = UUID().uuidString
}

func performUserOperation() async {
    await TaskLocals.$userID.withValue("user123") {
        await TaskLocals.$requestID.withValue("req456") {
            await someOperation()
            // userID and requestID are available here
        }
    }
}
```

### Key Points:

**• Task represents a unit of asynchronous work**
**• TaskGroup enables structured concurrency for multiple operations**
**• Tasks can be cancelled cooperatively**
**• Task-local values provide context inheritance**

### Notes:

*Tasks provide structured concurrency, making concurrent code predictable and manageable.*

## 4.3 Actors

Actors provide data isolation and protect against data races in concurrent programming.

### Code Example:

```
// Basic actor
actor Counter {
    private var value = 0

    func increment() -> Int {
        value += 1
        return value
    }

    func getValue() -> Int {
        return value
    }

    func reset() {
        value = 0
    }
}

// Using actors
func useCounter() async {
    let counter = Counter()

    let value1 = await counter.increment() // Must use await
    let value2 = await counter.getValue()

    print("Counter values: \(value1), \(value2)")
}
```

```swift
// Actor with async methods
actor ImageCache {
    private var images: [URL: UIImage] = [:]

    func image(for url: URL) async -> UIImage? {
        if let cached = images[url] {
            return cached
        }

        // Fetch image
        do {
            let (data, _) = try await URLSession.shared.data(from: url)
            let image = UIImage(data: data)
            images[url] = image
            return image
        } catch {
            return nil
        }
    }

    func clearCache() {
        images.removeAll()
    }
}

// MainActor for UI updates
@MainActor
class ViewModel: ObservableObject {
    @Published var data: [String] = []

    func loadData() async {
        let newData = await fetchDataFromNetwork()

        // This runs on main actor automatically
        self.data = newData
    }

    // Non-isolated methods can be called from any context
    nonisolated func validateInput(_ input: String) -> Bool {
        return !input.isEmpty
    }
}

// Global actor
@globalActor
actor DatabaseActor {
    static let shared = DatabaseActor()
    private init() {}
}

@DatabaseActor
func saveToDatabase(_ data: Data) {
    // All calls to this function are serialized
    // through the DatabaseActor
}

// Actor inheritance (only from protocols)
```

```
protocol Drawable {
    func draw() async
}

actor DrawingCanvas: Drawable {
    private var shapes: [Shape] = []

    func draw() async {
        for shape in shapes {
            await shape.render()
        }
    }

    func addShape(_ shape: Shape) {
        shapes.append(shape)
    }
}

// Sendable types for actor boundaries
struct SafeData: Sendable {
    let id: String
    let value: Int
}

actor DataProcessor {
    func process(_ data: SafeData) async -> ProcessedData {
        // Safe to pass Sendable types across actor boundaries
        return await processData(data)
    }
}
```

## Key Points:

**• Actors provide data isolation and prevent data races**
**• Actor methods are called with await from outside the actor**
**• MainActor ensures UI updates happen on the main thread**
**• Sendable types can be safely passed between actors**

## Notes:

*Actors are Swift's solution to thread-safe programming without explicit locks or queues.*

# PART III: SwiftUI FRAMEWORK

# Chapter 5: SwiftUI Fundamentals

## 5.1 Views and Modifiers

SwiftUI uses a declarative syntax where you describe what your UI should look like. Views are modified using modifiers that return new views.

### Code Example:

```
import SwiftUI

// Basic views
struct ContentView: View {
    var body: some View {
        VStack {
            Text("Hello, SwiftUI!")
                .font(.largeTitle)
                .foregroundColor(.blue)
                .padding()

            Image(systemName: "star.fill")
                .foregroundColor(.yellow)
                .font(.system(size: 50))

            Button("Tap Me") {
                print("Button tapped!")
            }
            .padding()
            .background(Color.blue)
            .foregroundColor(.white)
            .cornerRadius(10)
        }
    }
}

// Custom view with modifiers
struct CustomCard: View {
    let title: String
    let subtitle: String

    var body: some View {
        VStack(alignment: .leading) {
            Text(title)
                .font(.headline)
                .fontWeight(.bold)

            Text(subtitle)
                .font(.subheadline)
                .foregroundColor(.gray)
        }
        .padding()
        .background(Color.white)
        .cornerRadius(10)
        .shadow(radius: 5)
    }
}

// View composition
struct MainView: View {
    var body: some View {
        ScrollView {
            LazyVStack(spacing: 16) {
                CustomCard(title: "SwiftUI", subtitle: "Declarative UI framework")
                CustomCard(title: "Combine", subtitle: "Reactive programming")
                CustomCard(title: "Swift", subtitle: "Programming language")
            }
            .padding()
        }
    }
}

// ViewBuilder and conditional views
struct ConditionalView: View {
    @State private var showDetails = false

    var body: some View {
        VStack {
            Text("Main Content")

            if showDetails {
                Text("Additional Details")
                    .transition(.opacity)
            }

            Button(showDetails ? "Hide" : "Show") {
                withAnimation {
                    showDetails.toggle()
                }
            }
        }
    }
}
```

## Key Points:

**• Views are value types that describe UI declaratively**
**• Modifiers return new views, enabling method chaining**
**• View composition creates reusable components**
**• ViewBuilder enables conditional and loop-based view construction**

## Notes:

*SwiftUI's declarative approach means you describe the desired end state, and SwiftUI figures out how to get there.*

## 5.3 State Management

SwiftUI uses various property wrappers to manage state and data flow in your application.

### Code Example:

```swift
import SwiftUI

// @State for local state
struct CounterView: View {
    @State private var count = 0

    var body: some View {
        VStack {
            Text("Count: \(count)")
                .font(.largeTitle)

            HStack {
                Button("Increment") {
                    count += 1
                }

                Button("Decrement") {
                    count -= 1
                }

                Button("Reset") {
                    count = 0
                }
            }
        }
    }
}

// @Binding for two-way data flow
struct ToggleView: View {
    @Binding var isOn: Bool

    var body: some View {
        Toggle("Feature Enabled", isOn: $isOn)
            .padding()
    }
}

struct ParentView: View {
    @State private var featureEnabled = false

    var body: some View {
        VStack {
            Text("Feature is \(featureEnabled ? "ON" : "OFF")")
            ToggleView(isOn: $featureEnabled)
        }
    }
}

// @ObservableObject and @Published
```

```swift
class UserStore: ObservableObject {
    @Published var users: [User] = []
    @Published var isLoading = false
    @Published var errorMessage: String?

    func loadUsers() async {
        isLoading = true
        errorMessage = nil

        do {
            let fetchedUsers = try await NetworkService.fetchUsers()
            await MainActor.run {
                self.users = fetchedUsers
                self.isLoading = false
            }
        } catch {
            await MainActor.run {
                self.errorMessage = error.localizedDescription
                self.isLoading = false
            }
        }
    }
}

struct UserListView: View {
    @StateObject private var userStore = UserStore()

    var body: some View {
        NavigationView {
            Group {
                if userStore.isLoading {
                    ProgressView("Loading users...")
                } else if let error = userStore.errorMessage {
                    Text("Error: \(error)")
                        .foregroundColor(.red)
                } else {
                    List(userStore.users) { user in
                        UserRowView(user: user)
                    }
                }
            }
            .navigationTitle("Users")
            .task {
                await userStore.loadUsers()
            }
        }
    }
}

// @EnvironmentObject for dependency injection
struct ContentView: View {
    var body: some View {
        TabView {
            UserListView()
                .tabItem {
                    Image(systemName: "person.3")
                    Text("Users")
                }

            SettingsView()
                .tabItem {
                    Image(systemName: "gear")
                    Text("Settings")
                }
        }
        .environmentObject(UserStore())
    }
}

// @AppStorage for UserDefaults
struct SettingsView: View {
    @AppStorage("username") private var username = ""
    @AppStorage("isDarkMode") private var isDarkMode = false
    @AppStorage("fontSize") private var fontSize = 16.0

    var body: some View {
        Form {
            Section("User Preferences") {
                TextField("Username", text: $username)

                Toggle("Dark Mode", isOn: $isDarkMode)

                Stepper("Font Size: \(Int(fontSize))", value: $fontSize, in: 12...24)
            }
        }
        .preferredColorScheme(isDarkMode ? .dark : .light)
    }
}
```

### Key Points:

- **@State manages local view state**
- **@Binding creates two-way data connections**
- **@StateObject creates and owns ObservableObject instances**
- **@EnvironmentObject shares data across the view hierarchy**
- **@AppStorage automatically syncs with UserDefaults**

### Notes:

*SwiftUI's reactive state management system automatically updates views when data changes.*

## 5.4 Navigation

SwiftUI provides various navigation patterns including NavigationView, NavigationLink, and programmatic navigation.

### Code Example:

```
import SwiftUI
```

```swift
// Basic Navigation
struct ContentView: View {
    var body: some View {
        NavigationView {
            List {
                NavigationLink("Profile", destination: ProfileView())
                NavigationLink("Settings", destination: SettingsView())
                NavigationLink("About", destination: AboutView())
            }
            .navigationTitle("Main Menu")
            .navigationBarTitleDisplayMode(.large)
        }
    }
}

// NavigationStack (iOS 16+)
struct ModernNavigationView: View {
    @State private var path = NavigationPath()

    var body: some View {
        NavigationStack(path: $path) {
            List {
                Button("Go to Detail") {
                    path.append("detail")
                }

                Button("Go to Settings") {
                    path.append("settings")
                }
            }
            .navigationDestination(for: String.self) { value in
                switch value {
                case "detail":
                    DetailView()
                case "settings":
                    SettingsView()
                default:
                    Text("Unknown destination")
                }
            }
            .navigationTitle("Navigation Stack")
        }
    }
}

// Programmatic navigation
class NavigationController: ObservableObject {
    @Published var isShowingDetail = false
    @Published var selectedUser: User?

    func showUserDetail(_ user: User) {
        selectedUser = user
        isShowingDetail = true
    }

    func dismissDetail() {
        isShowingDetail = false
        selectedUser = nil
    }
}

struct UserListView: View {
    @StateObject private var navigation = NavigationController()
    @State private var users: [User] = []

    var body: some View {
        NavigationView {
            List(users) { user in
                Button(user.name) {
                    navigation.showUserDetail(user)
                }
            }
            .navigationTitle("Users")
            .sheet(isPresented: $navigation.isShowingDetail) {
                if let user = navigation.selectedUser {
                    UserDetailView(user: user)
                }
            }
        }
    }
}

// Tab Navigation
struct MainTabView: View {
    @State private var selectedTab = 0

    var body: some View {
        TabView(selection: $selectedTab) {
            HomeView()
                .tabItem {
                    Image(systemName: "house")
                    Text("Home")
                }
                .tag(0)

            SearchView()
                .tabItem {
                    Image(systemName: "magnifyingglass")
                    Text("Search")
                }
                .tag(1)

            ProfileView()
                .tabItem {
                    Image(systemName: "person")
                    Text("Profile")
                }
                .tag(2)
        }
        .accentColor(.blue)
    }
}

// Modal presentation
```

```
struct ModalExampleView: View {
    @State private var isShowingModal = false
    @State private var isShowingFullScreen = false

    var body: some View {
        VStack(spacing: 20) {
            Button("Show Sheet") {
                isShowingModal = true
            }
            .sheet(isPresented: $isShowingModal) {
                ModalContentView(isPresented: $isShowingModal)
            }

            Button("Show Full Screen") {
                isShowingFullScreen = true
            }
            .fullScreenCover(isPresented: $isShowingFullScreen) {
                FullScreenView(isPresented: $isShowingFullScreen)
            }
        }
    }
}

// Navigation with data passing
struct ProductListView: View {
    @State private var products: [Product] = []

    var body: some View {
        NavigationView {
            List(products) { product in
                NavigationLink(destination: ProductDetailView(product: product)) {
                    ProductRowView(product: product)
                }
            }
            .navigationTitle("Products")
            .toolbar {
                ToolbarItem(placement: .navigationBarTrailing) {
                    Button("Add") {
                        // Add new product
                    }
                }
            }
        }
    }
}
```

## Key Points:

**• NavigationView provides the foundation for navigation**
**• NavigationLink creates navigable connections between views**
**• NavigationStack (iOS 16+) offers more flexible navigation**
**• Sheet and fullScreenCover present modal views**
**• TabView creates tab-based navigation**

## Notes:

*SwiftUI navigation is declarative and integrates seamlessly with the framework's reactive patterns.*

## 5.5 Animations

SwiftUI provides powerful animation capabilities with simple, declarative syntax.

### Code Example:

```
import SwiftUI

// Basic animations
struct AnimationExamples: View {
    @State private var isRotated = false
    @State private var scale: CGFloat = 1.0
    @State private var offset: CGFloat = 0

    var body: some View {
        VStack(spacing: 40) {
            // Rotation animation
            Rectangle()
                .fill(Color.blue)
                .frame(width: 50, height: 50)
                .rotationEffect(.degrees(isRotated ? 180 : 0))
                .animation(.easeInOut(duration: 1), value: isRotated)
                .onTapGesture {
                    isRotated.toggle()
                }

            // Scale animation
            Circle()
                .fill(Color.green)
                .frame(width: 50, height: 50)
                .scaleEffect(scale)
                .animation(.spring(response: 0.5, dampingFraction: 0.6), value: scale)
                .onTapGesture {
                    scale = scale == 1.0 ? 1.5 : 1.0
                }

            // Offset animation
            RoundedRectangle(cornerRadius: 10)
                .fill(Color.orange)
                .frame(width: 50, height: 50)
                .offset(x: offset)
                .animation(.bouncy, value: offset)
                .onTapGesture {
                    offset = offset == 0 ? 100 : 0
                }
        }
    }
}

// withAnimation for explicit animation
```

```
struct ExplicitAnimationView: View {
    @State private var isExpanded = false

    var body: some View {
        VStack {
            RoundedRectangle(cornerRadius: 10)
                .fill(Color.purple)
                .frame(width: isExpanded ? 200 : 100, height: isExpanded ? 200 : 100)

            Button("Animate") {
                withAnimation(.spring(duration: 0.8)) {
                    isExpanded.toggle()
                }
            }
        }
    }
}

// Transitions
struct TransitionView: View {
    @State private var showDetail = false

    var body: some View {
        VStack {
            if showDetail {
                VStack {
                    Text("Detail View")
                        .font(.largeTitle)
                        .padding()

                    Text("This is additional detail information")
                        .padding()
                }
                .background(Color.gray.opacity(0.1))
                .cornerRadius(10)
                .transition(.asymmetric(
                    insertion: .move(edge: .trailing).combined(with: .opacity),
                    removal: .move(edge: .leading).combined(with: .opacity)
                ))
            }

            Button(showDetail ? "Hide" : "Show") {
                withAnimation(.easeInOut) {
                    showDetail.toggle()
                }
            }
        }
        .padding()
    }
}

// Custom animations
struct WaveView: View {
    @State private var animateWave = false

    var body: some View {
        ZStack {
            ForEach(0..<3) { index in
                Circle()
                    .stroke(Color.blue.opacity(0.3), lineWidth: 2)
                    .frame(width: 50, height: 50)
                    .scaleEffect(animateWave ? 3 : 1)
                    .opacity(animateWave ? 0 : 1)
                    .animation(.easeOut(duration: 2).repeatForever(autoreverses: false).dela
            }
        }
        .onAppear {
            animateWave = true
        }
    }
}

// Complex animation sequences
struct LoadingView: View {
    @State private var isLoading = false

    var body: some View {
        HStack {
            ForEach(0..<3) { index in
                Circle()
                    .fill(Color.blue)
                    .frame(width: 10, height: 10)
                    .scaleEffect(isLoading ? 1.0 : 0.5)
                    .animation(.easeInOut(duration: 0.6).repeatForever().delay(0.2 * Double(
            }
        }
        .onAppear {
            isLoading = true
        }
    }
}

// Gesture-driven animations
struct DragView: View {
    @State private var dragAmount = CGSize.zero

    var body: some View {
        VStack {
            RoundedRectangle(cornerRadius: 10)
                .fill(Color.red)
                .frame(width: 100, height: 100)
                .offset(dragAmount)
                .gesture(
                    DragGesture()
                        .onChanged { dragAmount = $0.translation }
                        .onEnded { _ in
                            withAnimation(.spring()) {
                                dragAmount = .zero
                            }
                        }
                )

            Text("Drag the square!")
                .padding()
            }
        }
    }
```

## Key Points:

**• Use .animation() modifier for implicit animations**
**• withAnimation provides explicit control over animations**
**• Transitions define how views appear and disappear**
**• Spring animations provide natural motion**
**• Combine animations with gestures for interactive experiences**

## Notes:

*SwiftUI animations are declarative and automatically handle the complex details of smooth transitions.*

# Chapter 6: Advanced SwiftUI

## 6.1 Custom Views and ViewModifiers

Create reusable custom views and view modifiers to build sophisticated UI components.

### Code Example:

```
import SwiftUI

// Custom View Components
struct PrimaryButton: View {
    let title: String
    let action: () -> Void
    @State private var isPressed = false

    var body: some View {
        Button(action: action) {
            Text(title)
                .font(.headline)
                .foregroundColor(.white)
                .frame(maxWidth: .infinity)
                .padding()
                .background(
                    RoundedRectangle(cornerRadius: 12)
                        .fill(Color.blue)
                        .scaleEffect(isPressed ? 0.95 : 1.0)
                )
        }
        .buttonStyle(PlainButtonStyle())
        .onLongPressGesture(minimumDuration: 0, maximumDistance: .infinity, pressing: { pres
            withAnimation(.easeInOut(duration: 0.1)) {
                isPressed = pressing
            }
        }, perform: {})
    }
}

// Custom ViewModifier
struct CardModifier: ViewModifier {
    let cornerRadius: CGFloat
    let shadowRadius: CGFloat

    func body(content: Content) -> some View {
        content
            .background(Color(.systemBackground))
            .cornerRadius(cornerRadius)
            .shadow(color: Color.black.opacity(0.1), radius: shadowRadius, x: 0, y: 2)
            .padding(.horizontal)
    }
}
```

```swift
extension View {
    func cardStyle(cornerRadius: CGFloat = 12, shadowRadius: CGFloat = 8) -> some View {
        self.modifier(CardModifier(cornerRadius: cornerRadius, shadowRadius: shadowRadius))
    }
}

// Custom Shape
struct CurvedRectangle: Shape {
    var cornerRadius: CGFloat
    var curveHeight: CGFloat

    func path(in rect: CGRect) -> Path {
        var path = Path()

        path.move(to: CGPoint(x: 0, y: curveHeight))
        path.addQuadCurve(to: CGPoint(x: rect.width, y: curveHeight),
                         control: CGPoint(x: rect.width / 2, y: 0))
        path.addLine(to: CGPoint(x: rect.width, y: rect.height - cornerRadius))
        path.addQuadCurve(to: CGPoint(x: rect.width - cornerRadius, y: rect.height),
                         control: CGPoint(x: rect.width, y: rect.height))
        path.addLine(to: CGPoint(x: cornerRadius, y: rect.height))
        path.addQuadCurve(to: CGPoint(x: 0, y: rect.height - cornerRadius),
                         control: CGPoint(x: 0, y: rect.height))
        path.closeSubpath()

        return path
    }
}

// Custom Progress View
struct CircularProgressView: View {
    let progress: Double
    let lineWidth: CGFloat = 8

    var body: some View {
        ZStack {
            Circle()
                .stroke(Color.gray.opacity(0.3), lineWidth: lineWidth)

            Circle()
                .trim(from: 0, to: progress)
                .stroke(
                    AngularGradient(colors: [.blue, .purple], center: .center),
                    style: StrokeStyle(lineWidth: lineWidth, lineCap: .round)
                )
                .rotationEffect(.degrees(-90))
                .animation(.easeInOut, value: progress)

            Text("\(Int(progress * 100))%")
                .font(.headline)
                .fontWeight(.semibold)
        }
    }
}

// Usage examples
struct CustomViewExamples: View {
    @State private var progress: Double = 0.0

    var body: some View {
        ScrollView {
            VStack(spacing: 20) {
                // Custom button
                PrimaryButton(title: "Custom Button") {
                    print("Button tapped!")
                }

                // Custom card view
                VStack {
                    Text("Card Content")
                        .font(.headline)
                    Text("This content uses the custom card modifier")
                        .font(.body)
                        .multilineTextAlignment(.center)
                }
                .cardStyle()

                // Custom shape
                CurvedRectangle(cornerRadius: 20, curveHeight: 30)
                    .fill(LinearGradient(colors: [.orange, .red], startPoint: .leading, endP
                    .frame(height: 100)

                // Custom progress view
                CircularProgressView(progress: progress)
                    .frame(width: 100, height: 100)

                Button("Update Progress") {
                    withAnimation {
                        progress = Double.random(in: 0...1)
                    }
                }
            }
            .padding()
        }
    }
}
```

### Key Points:

**• Custom views encapsulate reusable UI components**
**• ViewModifiers provide reusable styling and behavior**
**• Custom shapes enable unique visual designs**
**• Extensions make custom modifiers easy to use**

### Notes:

*Custom views and modifiers promote code reuse and maintainable UI architecture in SwiftUI.*

## 6.2 Gesture Handling

SwiftUI provides powerful gesture recognition for creating interactive user experiences.

## Code Example:

```
import SwiftUI

struct GestureExamples: View {
    @State private var offset = CGSize.zero
    @State private var scale: CGFloat = 1.0
    @State private var rotation: Angle = .degrees(0)
    @State private var longPressCount = 0

    var body: some View {
        ScrollView {
            VStack(spacing: 40) {
                // Drag Gesture
                VStack {
                    Text("Drag Gesture")
                        .font(.headline)

                    RoundedRectangle(cornerRadius: 10)
                        .fill(Color.blue)
                        .frame(width: 100, height: 100)
                        .offset(offset)
                        .gesture(
                            DragGesture()
                                .onChanged { value in
                                    offset = value.translation
                                }
                                .onEnded { _ in
                                    withAnimation(.spring()) {
                                        offset = .zero
                                    }
                                }
                        )
                }

                // Magnification Gesture
                VStack {
                    Text("Pinch to Scale")
                        .font(.headline)

                    Circle()
                        .fill(Color.green)
                        .frame(width: 80, height: 80)
                        .scaleEffect(scale)
                        .gesture(
                            MagnificationGesture()
                                .onChanged { value in
                                    scale = value
                                }
                                .onEnded { _ in
                                    withAnimation(.spring()) {
                                        scale = 1.0
                                    }
                                }
                        )
                }

                // Rotation Gesture
                VStack {
                    Text("Rotation Gesture")
                        .font(.headline)

                    Rectangle()
                        .fill(Color.orange)
                        .frame(width: 100, height: 60)
                        .rotationEffect(rotation)
                        .gesture(
                            RotationGesture()
                                .onChanged { value in
                                    rotation = value
                                }
                                .onEnded { _ in
                                    withAnimation(.spring()) {
                                        rotation = .degrees(0)
                                    }
                                }
                        )
                }

                // Long Press Gesture
                VStack {
                    Text("Long Press Count: \(longPressCount)")
                        .font(.headline)

                    RoundedRectangle(cornerRadius: 10)
                        .fill(Color.purple)
                        .frame(width: 120, height: 60)
                        .overlay(
                            Text("Long Press")
                                .foregroundColor(.white)
                                .font(.caption)
                        )
                        .onLongPressGesture(minimumDuration: 1.0) {
                            longPressCount += 1
                        }
                }

                // Combined Gestures
                VStack {
                    Text("Combined Gestures")
                        .font(.headline)

                    RoundedRectangle(cornerRadius: 15)
                        .fill(Color.red)
                        .frame(width: 100, height: 100)
                        .scaleEffect(scale)
                        .rotationEffect(rotation)
                        .offset(offset)
                        .gesture(
                            SimultaneousGesture(
                                DragGesture()
```

```
                                    .onChanged { value in
                                        offset = value.translation
                                    },
                                MagnificationGesture()
                                    .onChanged { value in
                                        scale = value
                                    }
                            )
                        )
                }

                Button("Reset All") {
                    withAnimation(.spring()) {
                        offset = .zero
                        scale = 1.0
                        rotation = .degrees(0)
                    }
                }
            }
            .padding()
        }
    }
}

// Custom gesture for swipe detection
struct SwipeGestureExample: View {
    @State private var swipeDirection: String = "None"

    var body: some View {
        VStack {
            Text("Swipe Direction: \(swipeDirection)")
                .font(.headline)
                .padding()

            Rectangle()
                .fill(Color.gray.opacity(0.3))
                .frame(width: 200, height: 200)
                .overlay(
                    Text("Swipe Me")
                        .font(.title)
                )
                .gesture(
                    DragGesture(minimumDistance: 50)
                        .onEnded { value in
                            let horizontalAmount = value.translation.x
                            let verticalAmount = value.translation.y

                            if abs(horizontalAmount) > abs(verticalAmount) {
                                swipeDirection = horizontalAmount < 0 ? "Left" : "Right"
                            } else {
                                swipeDirection = verticalAmount < 0 ? "Up" : "Down"
                            }
                        }
                )
        }
    }
}
```

### Key Points:

**• Drag, magnification, and rotation gestures provide intuitive interaction**
**• Long press gestures enable context-sensitive actions**
**• SimultaneousGesture combines multiple gestures**
**• Custom gesture logic enables app-specific interactions**

### Notes:

*SwiftUI gestures make apps feel responsive and natural to use across all Apple platforms.*

## 6.3 Performance Optimization

Techniques for optimizing SwiftUI app performance and responsiveness.

### Code Example:

```
import SwiftUI

// Lazy loading with LazyVStack and LazyHStack
struct LazyLoadingExample: View {
    let items = Array(1...10000)

    var body: some View {
        ScrollView {
            LazyVStack(spacing: 8) {
                ForEach(items, id: \.self) { item in
                    ExpensiveView(number: item)
                        .onAppear {
                            // Only create view when it appears
                            print("View \(item) appeared")
                        }
                }
            }
            .padding()
        }
    }
}

// Expensive view that should be lazy loaded
struct ExpensiveView: View {
    let number: Int

    var body: some View {
        HStack {
            // Simulate expensive operation
            Text("Item #\(number)")
            Spacer()
            Text(String(repeating: "•", count: Int.random(in: 1...5)))
        }
        .padding()
        .background(Color.blue.opacity(0.1))
```

```swift
            .cornerRadius(8)
        }
    }

    // Using @State vs @StateObject properly
    class ExpensiveDataModel: ObservableObject {
        @Published var data: [String] = []

        init() {
            // Expensive initialization
            loadData()
        }

        private func loadData() {
            // Simulate expensive data loading
            data = Array(1...1000).map { "Item \($0)" }
        }
    }

    struct OptimizedStateExample: View {
        // Use @StateObject for owned objects
        @StateObject private var dataModel = ExpensiveDataModel()

        // Use @State for simple values
        @State private var searchText = ""

        var filteredData: [String] {
            if searchText.isEmpty {
                return dataModel.data
            }
            return dataModel.data.filter { $0.localizedCaseInsensitiveContains(searchText) }
        }

        var body: some View {
            NavigationView {
                VStack {
                    SearchBar(text: $searchText)

                    List(filteredData, id: \.self) { item in
                        Text(item)
                    }
                }
                .navigationTitle("Optimized List")
            }
        }
    }

    // Efficient list updates with identifiable data
    struct IdentifiableDataExample: View {
        @State private var users: [User] = []

        var body: some View {
            List {
                ForEach(users) { user in
                    UserRowView(user: user)
                        .id(user.id) // Explicit ID for efficient updates
                }
                .onDelete(perform: deleteUsers)
            }
            .onAppear {
                loadUsers()
            }
        }

        private func deleteUsers(at offsets: IndexSet) {
            users.remove(atOffsets: offsets)
        }

        private func loadUsers() {
            // Load users efficiently
            users = UserService.loadUsers()
        }
    }

    // Using PreferenceKey for efficient data passing up the view hierarchy
    struct ScrollOffsetPreferenceKey: PreferenceKey {
        static var defaultValue: CGFloat = 0

        static func reduce(value: inout CGFloat, nextValue: () -> CGFloat) {
            value = nextValue()
        }
    }

    struct ScrollOffsetReader: View {
        @State private var scrollOffset: CGFloat = 0

        var body: some View {
            ScrollView {
                LazyVStack {
                    ForEach(0..<50) { index in
                        Text("Row \(index)")
                            .frame(maxWidth: .infinity)
                            .padding()
                            .background(Color.gray.opacity(0.1))
                    }
                }
                .background(
                    GeometryReader { geometry in
                        Color.clear
                            .preference(key: ScrollOffsetPreferenceKey.self,
                                      value: geometry.frame(in: .named("scrollView")).minY)
                    }
                )
            }
            .coordinateSpace(name: "scrollView")
            .onPreferenceChange(ScrollOffsetPreferenceKey.self) { value in
                scrollOffset = value
            }
            .overlay(
                Text("Offset: \(Int(scrollOffset))")
                    .padding()
                    .background(Color.black.opacity(0.7))
                    .foregroundColor(.white)
                    .cornerRadius(8)
                    .padding(.top),
                alignment: .topTrailing
            )
        }
    }

    // Memory-efficient image loading
    struct AsyncImageExample: View {
        let imageURL: URL

        var body: some View {
            AsyncImage(url: imageURL) { image in
                image
                    .resizable()
                    .aspectRatio(contentMode: .fill)
            } placeholder: {
                RoundedRectangle(cornerRadius: 10)
                    .fill(Color.gray.opacity(0.3))
                    .overlay(
                        ProgressView()
                            .scaleEffect(0.5)
                    )
            }
            .frame(width: 150, height: 150)
            .clipped()
            .cornerRadius(10)
        }
    }
```

### Key Points:

**• Use LazyVStack/LazyHStack for large lists to improve performance**
**• Choose @State vs @StateObject appropriately**
**• Provide explicit IDs for efficient list updates**
**• Use PreferenceKey for efficient upward data flow**
**• AsyncImage provides memory-efficient image loading**

### Notes:

*Performance optimization in SwiftUI focuses on lazy loading, proper state management, and efficient data flow.*

## 5.2 Layout System

SwiftUI provides powerful layout containers like VStack, HStack, ZStack, and LazyGrids for organizing views.

### Code Example:

```
import SwiftUI

// Basic stacks
struct LayoutExamples: View {
    var body: some View {
        VStack(spacing: 20) {
            // Horizontal stack
            HStack {
                Text("Left")
                Spacer()
                Text("Right")
            }
            .padding()
            .background(Color.gray.opacity(0.2))

            // Vertical stack with alignment
            VStack(alignment: .leading, spacing: 10) {
                Text("Title")
                    .font(.headline)
                Text("This is a longer subtitle that demonstrates alignment")
                    .font(.caption)
            }
            .frame(maxWidth: .infinity, alignment: .leading)
            .padding()
            .background(Color.blue.opacity(0.1))

            // Overlay stack
            ZStack {
                Rectangle()
                    .fill(Color.orange)
                    .frame(width: 100, height: 100)

                Text("Overlay")
                    .foregroundColor(.white)
                    .font(.caption)
            }
        }
    }
}

// Grid layouts
struct GridExample: View {
    let items = Array(1...20)

    let columns = [
        GridItem(.flexible()),
        GridItem(.flexible()),
        GridItem(.flexible())
    ]

    var body: some View {
        ScrollView {
            LazyVGrid(columns: columns, spacing: 10) {
                ForEach(items, id: \.self) { item in
                    RoundedRectangle(cornerRadius: 8)
                        .fill(Color.blue)
                        .frame(height: 50)
                        .overlay(
                            Text("\(item)")
                                .foregroundColor(.white)
                        )
                }
            }
            .padding()
        }
    }
}

// Adaptive grids
struct AdaptiveGridExample: View {
    let items = Array(1...50)

    var body: some View {
        ScrollView {
            LazyVGrid(columns: [GridItem(.adaptive(minimum: 80))], spacing: 10) {
                ForEach(items, id: \.self) { item in
                    Circle()
                        .fill(Color.green)
                        .frame(height: 80)
                        .overlay(
                            Text("\(item)")
                                .foregroundColor(.white)
                        )
                }
            }
            .padding()
        }
    }
}

// GeometryReader for custom layouts
struct CustomLayoutView: View {
    var body: some View {
        GeometryReader { geometry in
            VStack {
                Rectangle()
                    .fill(Color.red)
                    .frame(width: geometry.size.width * 0.8, height: 50)

                HStack {
                    Rectangle()
                        .fill(Color.blue)
                        .frame(width: geometry.size.width * 0.4, height: 100)

                    Spacer()

                    Rectangle()
                        .fill(Color.green)
                        .frame(width: geometry.size.width * 0.4, height: 100)
                }
            }
            .frame(width: geometry.size.width, height: geometry.size.height)
        }
    }
}
```

## Key Points:

**• VStack, HStack, and ZStack are the fundamental layout containers**
**• Spacer() pushes views apart or centers them**
**• LazyVGrid and LazyHGrid create efficient grid layouts**
**• GeometryReader provides access to parent view dimensions**

## Notes:

*SwiftUI's layout system is designed to be predictable and easy to understand while being highly flexible.*

# PART IV: REACTIVE PROGRAMMING

# Chapter 7: Combine Framework

## 7.1 Publishers and Subscribers

Combine is Apple's framework for handling asynchronous events by combining event-processing operators. Publishers emit values over time, and subscribers receive them.

### Code Example:

```
import Combine
import Foundation

// Basic publisher and subscriber
class CombineBasics {
    var cancellables = Set<AnyCancellable>()

    func basicPublisherSubscriber() {
        // Simple publisher
        let publisher = Just("Hello, Combine!")

        publisher
            .sink { value in
                print("Received: \(value)")
            }
            .store(in: &cancellables)

        // Array publisher
        let numbers = [1, 2, 3, 4, 5]
        numbers.publisher
            .sink { number in
                print("Number: \(number)")
            }
            .store(in: &cancellables)
    }

    // PassthroughSubject
    func passthroughSubjectExample() {
        let subject = PassthroughSubject<String, Never>()

        subject
            .sink { value in
                print("PassthroughSubject received: \(value)")
            }
            .store(in: &cancellables)

        subject.send("First message")
        subject.send("Second message")
        subject.send(completion: .finished)
    }

    // CurrentValueSubject
func currentValueSubjectExample() {
    let currentValueSubject = CurrentValueSubject<Int, Never>(0)

    currentValueSubject
        .sink { value in
            print("CurrentValueSubject: \(value)")
        }
        .store(in: &cancellables)

    currentValueSubject.send(1)
    currentValueSubject.send(2)

    print("Current value: \(currentValueSubject.value)")
}

// Custom publisher
struct CountdownPublisher: Publisher {
    typealias Output = Int
    typealias Failure = Never

    let start: Int

    func receive<S>(subscriber: S) where S : Subscriber, Never == S.Failure, Int == S.In
        let subscription = CountdownSubscription(subscriber: subscriber, start: start)
        subscriber.receive(subscription: subscription)
    }
}

class CountdownSubscription<S: Subscriber>: Subscription where S.Input == Int, S.Failure
    private var subscriber: S?
    private var current: Int

    init(subscriber: S, start: Int) {
        self.subscriber = subscriber
        self.current = start
    }

    func request(_ demand: Subscribers.Demand) {
        var demand = demand

        while demand > 0 && current > 0 {
            _ = subscriber?.receive(current)
            current -= 1
            demand -= 1
        }

        if current == 0 {
            subscriber?.receive(completion: .finished)
        }
    }

    func cancel() {
        subscriber = nil
    }
}

func customPublisherExample() {
        CountdownPublisher(start: 5)
            .sink { value in
                print("Countdown: \(value)")
            }
            .store(in: &cancellables)
    }
}
```

### Key Points:

- **Publishers emit values over time, subscribers receive them**
- **PassthroughSubject sends values to subscribers without storing current value**
- **CurrentValueSubject maintains and emits the current value to new subscribers**
- **Custom publishers implement the Publisher protocol**

### Notes:

*Combine follows the reactive programming paradigm, making asynchronous code more manageable and composable.*

## 7.2 Combine Operators

Combine provides dozens of operators for transforming, filtering, and combining publisher streams.

### Code Example:

```
import Combine
import Foundation

class CombineOperatorsExample: ObservableObject {
    var cancellables = Set<AnyCancellable>()

    func transformationOperators() {
        // Map - transform each element
        [1, 2, 3, 4, 5].publisher
            .map { $0 * 2 }
            .sink { print("Doubled: \($0)") }
            .store(in: &cancellables)

        // FlatMap - flatten nested publishers
        ["apple", "banana", "cherry"].publisher
            .flatMap { fruit in
                Just(fruit.uppercased())
                    .delay(for: .seconds(1), scheduler: RunLoop.main)
            }
            .sink { print("Uppercased: \($0)") }
            .store(in: &cancellables)

        // CompactMap - filter out nil values
        ["1", "2", "three", "4", "five"].publisher
            .compactMap { Int($0) }
            .sink { print("Valid number: \($0)") }
            .store(in: &cancellables)
        // Scan - accumulate values
        [1, 2, 3, 4, 5].publisher
            .scan(0, +)
            .sink { print("Running sum: \($0)") }
            .store(in: &cancellables)
    }

    func filteringOperators() {
        // Filter - include only matching elements
        (1...10).publisher
            .filter { $0 % 2 == 0 }
            .sink { print("Even number: \($0)") }
            .store(in: &cancellables)

        // RemoveDuplicates - filter consecutive duplicates
        [1, 1, 2, 2, 2, 3, 3, 4, 4, 4, 4].publisher
            .removeDuplicates()
            .sink { print("Unique: \($0)") }
            .store(in: &cancellables)

        // DropFirst/DropLast - skip elements
        [1, 2, 3, 4, 5, 6, 7, 8, 9, 10].publisher
            .dropFirst(3)
            .dropLast(2)
            .sink { print("Middle values: \($0)") }
            .store(in: &cancellables)

        // Prefix - take only first n elements
        (1...100).publisher
            .prefix(5)
            .sink { print("First 5: \($0)") }
            .store(in: &cancellables)
    }

    func combiningOperators() {
        let publisher1 = PassthroughSubject<String, Never>()
        let publisher2 = PassthroughSubject<String, Never>()

        // Merge - combine multiple publishers
        Publishers.Merge(publisher1, publisher2)
            .sink { print("Merged: \($0)") }
            .store(in: &cancellables)

        // CombineLatest - emit when any publisher emits
        Publishers.CombineLatest(publisher1, publisher2)
            .sink { value1, value2 in
                print("Combined: \(value1) + \(value2)")
            }
            .store(in: &cancellables)

        // Zip - pair corresponding elements
        let numbers = [1, 2, 3, 4, 5].publisher
        let letters = ["A", "B", "C", "D", "E"].publisher

        Publishers.Zip(numbers, letters)
            .sink { number, letter in
                print("Zipped: \(number)\(letter)")
            }
            .store(in: &cancellables)
    }

    func timingOperators() {
        // Debounce - wait for pause in emissions
        let searchSubject = PassthroughSubject<String, Never>()

        searchSubject
            .debounce(for: .milliseconds(500), scheduler: RunLoop.main)
            .removeDuplicates()
            .sink { searchTerm in
                print("Searching for: \(searchTerm)")
                // Perform search here
            }
            .store(in: &cancellables)

        // Throttle - limit emission frequency
        Timer.publish(every: 0.1, on: .main, in: .common)
            .autoconnect()
            .throttle(for: .seconds(1), scheduler: RunLoop.main, latest: true)
            .sink { date in
                print("Throttled timer: \(date)")
            }
            .store(in: &cancellables)

        // Delay - delay emissions
        ["Immediate", "Delayed"].publisher
            .delay(for: .seconds(2), scheduler: DispatchQueue.main)
            .sink { print("After delay: \($0)") }
            .store(in: &cancellables)

        // Timeout - fail if no emission within time limit
        Just("Hello")
            .delay(for: .seconds(3), scheduler: DispatchQueue.main)
            .timeout(.seconds(2), scheduler: DispatchQueue.main)
            .sink(
                receiveCompletion: { completion in
                    switch completion {
                    case .failure:
                        print("Timed out!")
                    case .finished:
                        print("Completed")
                    }
                },
                receiveValue: { print("Received: \($0)") }
            )
            .store(in: &cancellables)
    }

    func errorHandlingOperators() {
        enum NetworkError: Error {
            case connectionFailed
            case timeout
        }

        // Catch - handle errors and provide fallback
```

```
        Fail<String, NetworkError>(error: .connectionFailed)
            .catch { error in
                Just("Fallback value")
            }
            .sink { print("Result: \($0)") }
            .store(in: &cancellables)

        // Retry - retry failed operations
        let failingPublisher = PassthroughSubject<String, NetworkError>()

        failingPublisher
            .retry(3)
            .sink(
                receiveCompletion: { completion in
                    print("Final completion: \(completion)")
                },
                receiveValue: { print("Value: \($0)") }
            )
            .store(in: &cancellables)

        // ReplaceError - replace errors with a value
        Fail<String, NetworkError>(error: .timeout)
            .replaceError(with: "Default response")
            .sink { print("Error replaced with: \($0)") }
            .store(in: &cancellables)
    }
}
```

### Key Points:

- **Map, flatMap, and compactMap transform publisher values**
- **Filter, removeDuplicates control which values pass through**
- **Merge, combineLatest, and zip combine multiple publishers**
- **Debounce, throttle, and delay control timing of emissions**
- **Catch, retry, and replaceError handle failure scenarios**

### Notes:

*Combine operators provide a declarative way to process asynchronous data streams with powerful composition capabilities.*

## 7.3 Networking with Combine

Combine integrates seamlessly with URLSession for reactive networking and data processing.

### Code Example:

```
import Combine
import Foundation

// Network service using Combine
class NetworkService: ObservableObject {
    @Published var isLoading = false
```

```swift
@Published var users: [User] = []
@Published var errorMessage: String?

private var cancellables = Set<AnyCancellable>()

// Generic API request method
func request<T: Codable>(url: URL, type: T.Type) -> AnyPublisher<T, NetworkError> {
    URLSession.shared.dataTaskPublisher(for: url)
        .tryMap { data, response -> Data in
            guard let httpResponse = response as? HTTPURLResponse,
                  200...299 ~= httpResponse.statusCode else {
                throw NetworkError.invalidResponse
            }
            return data
        }
        .decode(type: type, decoder: JSONDecoder())
        .mapError { error in
            if error is DecodingError {
                return NetworkError.decodingFailed
            }
            return NetworkError.networkFailed
        }
        .receive(on: DispatchQueue.main)
        .eraseToAnyPublisher()
}

// Fetch users with error handling
func fetchUsers() {
    guard let url = URL(string: "https://jsonplaceholder.typicode.com/users") else {
        errorMessage = "Invalid URL"
        return
    }

    isLoading = true
    errorMessage = nil

    request(url: url, type: [User].self)
        .sink(
            receiveCompletion: { [weak self] completion in
                self?.isLoading = false
                switch completion {
                case .failure(let error):
                    self?.errorMessage = error.localizedDescription
                case .finished:
                    break
                }
            },
            receiveValue: { [weak self] users in
                self?.users = users
            }
        )
        .store(in: &cancellables)
}

// Search with debouncing
func searchUsers(query: String) -> AnyPublisher<[User], NetworkError> {
    guard let url = URL(string: "https://jsonplaceholder.typicode.com/users") else {
            return Fail(error: NetworkError.invalidURL)
                .eraseToAnyPublisher()
        }

        return Just(query)
            .debounce(for: .milliseconds(300), scheduler: RunLoop.main)
            .removeDuplicates()
            .flatMap { searchTerm in
                self.request(url: url, type: [User].self)
                    .map { users in
                        users.filter { user in
                            user.name.localizedCaseInsensitiveContains(searchTerm)
                        }
                    }
            }
            .eraseToAnyPublisher()
    }

    // Upload with progress tracking
    func uploadFile(data: Data, to url: URL) -> AnyPublisher<UploadResponse, NetworkError> {
        var request = URLRequest(url: url)
        request.httpMethod = "POST"
        request.setValue("application/json", forHTTPHeaderField: "Content-Type")
        request.httpBody = data

        return URLSession.shared.dataTaskPublisher(for: request)
            .tryMap { data, response -> Data in
                guard let httpResponse = response as? HTTPURLResponse,
                      200...299 ~= httpResponse.statusCode else {
                    throw NetworkError.serverError
                }
                return data
            }
            .decode(type: UploadResponse.self, decoder: JSONDecoder())
            .mapError { _ in NetworkError.uploadFailed }
            .receive(on: DispatchQueue.main)
            .eraseToAnyPublisher()
    }

    // Batch requests with error recovery
    func fetchMultipleResources() -> AnyPublisher<CombinedData, Never> {
        let usersPublisher = request(url: URL(string: "https://api.example.com/users")!, typ
            .catch { _ in Just([User]()) } // Fallback to empty array on error

        let postsPublisher = request(url: URL(string: "https://api.example.com/posts")!, typ
            .catch { _ in Just([Post]()) } // Fallback to empty array on error

        return Publishers.CombineLatest(usersPublisher, postsPublisher)
            .map { users, posts in
                CombinedData(users: users, posts: posts)
            }
            .eraseToAnyPublisher()
    }
}

// Error types for networking
enum NetworkError: Error, LocalizedError {
```

```swift
    case invalidURL
    case networkFailed
    case invalidResponse
    case decodingFailed
    case serverError
    case uploadFailed

    var errorDescription: String? {
        switch self {
        case .invalidURL:
            return "Invalid URL"
        case .networkFailed:
            return "Network request failed"
        case .invalidResponse:
            return "Invalid server response"
        case .decodingFailed:
            return "Failed to decode response"
        case .serverError:
            return "Server error occurred"
        case .uploadFailed:
            return "Upload failed"
        }
    }
}

// Usage in SwiftUI
struct NetworkingExampleView: View {
    @StateObject private var networkService = NetworkService()
    @State private var searchText = ""
    @State private var searchResults: [User] = []

    var body: some View {
        NavigationView {
            VStack {
                SearchBar(text: $searchText)
                    .onChange(of: searchText) { newValue in
                        searchUsers(query: newValue)
                    }

                if networkService.isLoading {
                    ProgressView("Loading...")
                        .frame(maxWidth: .infinity, maxHeight: .infinity)
                } else if let error = networkService.errorMessage {
                    Text("Error: \(error)")
                        .foregroundColor(.red)
                        .padding()
                } else {
                    List(searchResults.isEmpty ? networkService.users : searchResults) { use
                        VStack(alignment: .leading) {
                            Text(user.name)
                                .font(.headline)
                            Text(user.email)
                                .font(.caption)
                                .foregroundColor(.secondary)
                        }
                    }
                }
            }
            .navigationTitle("Users")
            .onAppear {
                networkService.fetchUsers()
            }
        }
    }

    private func searchUsers(query: String) {
        networkService.searchUsers(query: query)
            .sink(
                receiveCompletion: { _ in },
                receiveValue: { users in
                    searchResults = users
                }
            )
            .store(in: &networkService.cancellables)
    }
}
```

## Key Points:

**• URLSession dataTaskPublisher integrates with Combine**
**• Use tryMap for response validation and error handling**
**• Debounce search queries to reduce API calls**
**• Combine multiple API calls with CombineLatest or Zip**
**• Handle errors gracefully with catch and fallback values**

## Notes:

*Combine transforms networking from callback-based to reactive, making complex data flows more manageable.*

# PART VI: iOS DEVELOPMENT

# Chapter 9: Data Persistence

## 9.1 UserDefaults and AppStorage

UserDefaults provides simple key-value storage for user preferences and app settings.

### Code Example:

```
import SwiftUI
import Foundation

// UserDefaults wrapper for type safety
@propertyWrapper
struct UserDefault<T> {
    let key: String
    let defaultValue: T

    var wrappedValue: T {
        get {
            UserDefaults.standard.object(forKey: key) as? T ?? defaultValue
        }
        set {
            UserDefaults.standard.set(newValue, forKey: key)
        }
    }
}

// Settings manager using UserDefaults
class AppSettings: ObservableObject {
    @UserDefault(key: "username", defaultValue: "")
    var username: String

    @UserDefault(key: "isDarkMode", defaultValue: false)
    var isDarkMode: Bool

    @UserDefault(key: "fontSize", defaultValue: 16.0)
    var fontSize: Double

    @UserDefault(key: "notifications", defaultValue: true)
    var notificationsEnabled: Bool

    @UserDefault(key: "lastLoginDate", defaultValue: Date.distantPast)
    var lastLoginDate: Date

    // Complex data storage with Codable
    @UserDefault(key: "favoriteItems", defaultValue: [])
    var favoriteItems: [String]

    // Store custom objects
    var userProfile: UserProfile? {
        get {
            guard let data = UserDefaults.standard.data(forKey: "userProfile") else { return
            return try? JSONDecoder().decode(UserProfile.self, from: data)
        }
        set {
            let data = try? JSONEncoder().encode(newValue)
            UserDefaults.standard.set(data, forKey: "userProfile")
        }
    }

    func resetToDefaults() {
        username = ""
        isDarkMode = false
        fontSize = 16.0
        notificationsEnabled = true
        lastLoginDate = Date.distantPast
        favoriteItems = []
        userProfile = nil
    }
}

// SwiftUI integration with @AppStorage
struct SettingsView: View {
    @AppStorage("username") private var username: String = ""
    @AppStorage("isDarkMode") private var isDarkMode: Bool = false
    @AppStorage("fontSize") private var fontSize: Double = 16.0
    @AppStorage("theme") private var selectedTheme: Theme = .system

    var body: some View {
        NavigationView {
            Form {
                Section("Account") {
                    TextField("Username", text: $username)
                        .textContentType(.username)

                    Toggle("Enable Notifications", isOn: .constant(true))
                }

                Section("Appearance") {
                    Toggle("Dark Mode", isOn: $isDarkMode)

                    HStack {
                        Text("Font Size")
                        Spacer()
                        Text("\(Int(fontSize))pt")
                            .foregroundColor(.secondary)
                    }

                    Slider(value: $fontSize, in: 12...24, step: 1)

                    Picker("Theme", selection: $selectedTheme) {
                        ForEach(Theme.allCases, id: \.self) { theme in
                            Text(theme.displayName).tag(theme)
                        }
                    }
                }

                Section("Data") {
                    Button("Reset Settings", role: .destructive) {
                        resetSettings()
                    }

                    Button("Export Settings") {
                        exportSettings()
                    }
                }
            }
            .navigationTitle("Settings")
        }
        .preferredColorScheme(isDarkMode ? .dark : .light)
    }

    private func resetSettings() {
        username = ""
        isDarkMode = false
        fontSize = 16.0
        selectedTheme = .system
    }

    private func exportSettings() {
        let settings = [
            "username": username,
            "isDarkMode": isDarkMode,
            "fontSize": fontSize,
            "theme": selectedTheme.rawValue
        ]

        // Export logic here
        print("Exported settings: \(settings)")
    }
}

// Theme enum for UserDefaults
enum Theme: String, CaseIterable, Codable {
    case system = "system"
    case light = "light"
    case dark = "dark"

    var displayName: String {
        switch self {
        case .system: return "System"
        case .light: return "Light"
        case .dark: return "Dark"
        }
    }
}

// User profile model
struct UserProfile: Codable {
    let id: String
    let name: String
    let email: String
    let avatar: URL?
    let preferences: [String: String]
    static let example = UserProfile(
        id: UUID().uuidString,
        name: "John Doe",
        email: "john@example.com",
        avatar: URL(string: "https://example.com/avatar.jpg"),
        preferences: ["theme": "dark", "language": "en"]
    )
}

// Advanced UserDefaults usage
extension UserDefaults {
    func set<T: Codable>(_ object: T, forKey key: String) {
        let data = try? JSONEncoder().encode(object)
        set(data, forKey: key)
    }

    func get<T: Codable>(_ type: T.Type, forKey key: String) -> T? {
        guard let data = data(forKey: key) else { return nil }
        return try? JSONDecoder().decode(type, from: data)
    }

    func remove(forKey key: String) {
        removeObject(forKey: key)
    }
}
```

### Key Points:

- **UserDefaults provides simple persistent storage for app settings**
- **@AppStorage automatically syncs SwiftUI views with UserDefaults**
- **Property wrappers make UserDefaults type-safe and easy to use**
- **Store complex objects using Codable and JSON encoding**
- **Group related settings in dedicated classes for organization**

### Notes:

*UserDefaults is perfect for storing user preferences, app settings, and simple persistent data in iOS apps.*

## 9.2 Core Data Basics

Core Data provides object graph management and persistence for complex data models in iOS applications.

### Code Example:

```
import CoreData
import SwiftUI

// Core Data Stack
class CoreDataManager: ObservableObject {
    static let shared = CoreDataManager()
    lazy var persistentContainer: NSPersistentContainer = {
        let container = NSPersistentContainer(name: "DataModel")
        container.loadPersistentStores { _, error in
            if let error = error {
                fatalError("Core Data error: \(error)")
            }
        }
        return container
    }()

    var context: NSManagedObjectContext {
        persistentContainer.viewContext
    }

    func save() {
        let context = persistentContainer.viewContext

        if context.hasChanges {
            do {
                try context.save()
            } catch {
                print("Save error: \(error)")
            }
        }
    }
}

// Core Data Entity (User+CoreDataClass.swift)
@objc(User)
public class User: NSManagedObject {
    @nonobjc public class func fetchRequest() -> NSFetchRequest<User> {
        return NSFetchRequest<User>(entityName: "User")
    }

    @NSManaged public var id: UUID
    @NSManaged public var name: String
    @NSManaged public var email: String
    @NSManaged public var createdDate: Date
    @NSManaged public var posts: NSSet?

    // Computed properties
    public var postsArray: [Post] {
        let set = posts as? Set<Post> ?? []
        return set.sorted { $0.createdDate < $1.createdDate }
    }

    // Convenience initializer
    convenience init(context: NSManagedObjectContext, name: String, email: String) {
        self.init(context: context)
        self.id = UUID()
        self.name = name
        self.email = email
        self.createdDate = Date()
    }
}

// Repository pattern for Core Data operations
```

```
class UserRepository: ObservableObject {
    private let coreDataManager = CoreDataManager.shared
    @Published var users: [User] = []
    @Published var isLoading = false

    init() {
        fetchUsers()
    }

    func fetchUsers() {
        isLoading = true
        let request: NSFetchRequest<User> = User.fetchRequest()
        request.sortDescriptors = [NSSortDescriptor(keyPath: \User.name, ascending: true)]

        do {
            users = try coreDataManager.context.fetch(request)
        } catch {
            print("Fetch error: \(error)")
        }
        isLoading = false
    }

    func addUser(name: String, email: String) {
        let user = User(context: coreDataManager.context, name: name, email: email)
        coreDataManager.save()
        fetchUsers()
    }

    func deleteUser(_ user: User) {
        coreDataManager.context.delete(user)
        coreDataManager.save()
        fetchUsers()
    }

    func updateUser(_ user: User, name: String, email: String) {
        user.name = name
        user.email = email
        coreDataManager.save()
        fetchUsers()
    }

    func searchUsers(by name: String) -> [User] {
        let request: NSFetchRequest<User> = User.fetchRequest()
        request.predicate = NSPredicate(format: "name CONTAINS[cd] %@", name)
        request.sortDescriptors = [NSSortDescriptor(keyPath: \User.name, ascending: true)]

        do {
            return try coreDataManager.context.fetch(request)
        } catch {
            print("Search error: \(error)")
            return []
        }
    }
}

// SwiftUI integration with Core Data
struct UserListView: View {
```

```swift
    @StateObject private var repository = UserRepository()
    @State private var showingAddUser = false

    var body: some View {
        NavigationView {
            List {
                if repository.isLoading {
                    ProgressView("Loading users...")
                        .frame(maxWidth: .infinity)
                } else {
                    ForEach(repository.users, id: \.id) { user in
                        VStack(alignment: .leading) {
                            Text(user.name)
                                .font(.headline)
                            Text(user.email)
                                .font(.caption)
                                .foregroundColor(.secondary)
                            Text("Created: \(user.createdDate, style: .date)")
                                .font(.caption2)
                                .foregroundColor(.secondary)
                        }
                    }
                    .onDelete(perform: deleteUsers)
                }
            }
            .navigationTitle("Users")
            .toolbar {
                ToolbarItem(placement: .navigationBarTrailing) {
                    Button("Add") {
                        showingAddUser = true
                    }
                }
            }
            .sheet(isPresented: $showingAddUser) {
                AddUserView { name, email in
                    repository.addUser(name: name, email: email)
                }
            }
        }
    }

    private func deleteUsers(offsets: IndexSet) {
        for index in offsets {
            let user = repository.users[index]
            repository.deleteUser(user)
        }
    }
}

// Add user form
struct AddUserView: View {
    @Environment(\.dismiss) private var dismiss
    @State private var name = ""
    @State private var email = ""

    let onSave: (String, String) -> Void
```

```
        var body: some View {
            NavigationView {
                Form {
                    TextField("Name", text: $name)
                    TextField("Email", text: $email)
                        .textContentType(.emailAddress)
                        .keyboardType(.emailAddress)
                }
                .navigationTitle("Add User")
                .toolbar {
                    ToolbarItem(placement: .navigationBarLeading) {
                        Button("Cancel") {
                            dismiss()
                        }
                    }

                    ToolbarItem(placement: .navigationBarTrailing) {
                        Button("Save") {
                            onSave(name, email)
                            dismiss()
                        }
                        .disabled(name.isEmpty || email.isEmpty)
                    }
                }
            }
        }
    }
```

## Key Points:

**• Core Data provides object graph management and persistence**
**• Use NSPersistentContainer to set up the Core Data stack**
**• Repository pattern encapsulates Core Data operations**
**• NSFetchRequest with predicates enables complex queries**
**• SwiftUI integrates seamlessly with Core Data through ObservableObject**

## Notes:

*Core Data is ideal for complex data models with relationships and advanced querying capabilities.*

# Chapter 10: Testing & Architecture

## 10.1 Unit Testing

Unit testing ensures code reliability and maintainability through automated testing of individual components.

### Code Example:

```
import XCTest
@testable import MyApp

// Test class structure
class CalculatorTests: XCTestCase {
    var calculator: Calculator!

    override func setUpWithError() throws {
        // Set up test objects before each test
        calculator = Calculator()
    }

    override func tearDownWithError() throws {
        // Clean up after each test
        calculator = nil
    }

    // Basic test methods
    func testAddition() {
        let result = calculator.add(5, 3)
        XCTAssertEqual(result, 8, "5 + 3 should equal 8")
    }

    func testSubtraction() {
        let result = calculator.subtract(10, 4)
        XCTAssertEqual(result, 6)
    }

    func testDivision() {
        let result = calculator.divide(12, 3)
        XCTAssertEqual(result, 4)
    }

    func testDivisionByZero() {
        XCTAssertThrowsError(try calculator.divide(10, 0)) { error in
            XCTAssertEqual(error as? CalculatorError, CalculatorError.divisionByZero)
        }
    }
}

// Testing asynchronous code
class NetworkServiceTests: XCTestCase {
    var networkService: NetworkService!

    override func setUp() {
        networkService = NetworkService()
    }

    func testFetchUsers() async throws {
        let users = try await networkService.fetchUsers()
        XCTAssertGreaterThan(users.count, 0)
        XCTAssertNotNil(users.first?.name)
    }

    func testFetchUsersWithExpectation() {
        let expectation = XCTestExpectation(description: "Fetch users")

        networkService.fetchUsers { result in
            switch result {
            case .success(let users):
                XCTAssertGreaterThan(users.count, 0)
            case .failure(let error):
                XCTFail("Failed with error: \(error)")
            }
            expectation.fulfill()
        }

        wait(for: [expectation], timeout: 10.0)
    }
}

// Testing with mocks
protocol UserRepositoryProtocol {
    func fetchUsers() async throws -> [User]
    func saveUser(_ user: User) async throws
}

class MockUserRepository: UserRepositoryProtocol {
    var shouldThrowError = false
    var mockUsers: [User] = []

    func fetchUsers() async throws -> [User] {
        if shouldThrowError {
            throw NetworkError.connectionFailed
        }
        return mockUsers
    }

    func saveUser(_ user: User) async throws {
        if shouldThrowError {
            throw DatabaseError.saveFailed
        }
        mockUsers.append(user)
    }
}

class UserViewModelTests: XCTestCase {
    var viewModel: UserViewModel!
    var mockRepository: MockUserRepository!
    override func setUp() {
        mockRepository = MockUserRepository()
        viewModel = UserViewModel(repository: mockRepository)
    }

    func testLoadUsersSuccess() async {
        // Arrange
        let expectedUsers = [User(name: "John", email: "john@test.com")]
        mockRepository.mockUsers = expectedUsers

        // Act
        await viewModel.loadUsers()

        // Assert
        XCTAssertEqual(viewModel.users.count, 1)
        XCTAssertEqual(viewModel.users.first?.name, "John")
        XCTAssertFalse(viewModel.isLoading)
        XCTAssertNil(viewModel.errorMessage)
    }

    func testLoadUsersFailure() async {
        // Arrange
        mockRepository.shouldThrowError = true

        // Act
        await viewModel.loadUsers()

        // Assert
        XCTAssertTrue(viewModel.users.isEmpty)
        XCTAssertNotNil(viewModel.errorMessage)
        XCTAssertFalse(viewModel.isLoading)
    }
}

// Performance testing
class PerformanceTests: XCTestCase {
    func testSortingPerformance() {
        let largeArray = (1...10000).shuffled()

        measure {
            _ = largeArray.sorted()
        }
    }

    func testAsyncPerformance() async {
        await measure {
            await performExpensiveAsyncOperation()
        }
    }
}

// Test utilities
extension XCTestCase {
    func waitForAsync<T>(
        _ asyncFunction: @escaping () async throws -> T,
        timeout: TimeInterval = 5.0
    ) async throws -> T {
        return try await withCheckedThrowingContinuation { continuation in
            Task {
                do {
                    let result = try await asyncFunction()
                    continuation.resume(returning: result)
                } catch {
                    continuation.resume(throwing: error)
                }
            }
        }
    }
}
```

### Key Points:

- **XCTestCase provides the foundation for unit testing**
- **Use setUp/tearDown for test preparation and cleanup**
- **Mock objects isolate units under test**
- **XCTestExpectation handles asynchronous testing**
- **Performance tests measure code efficiency**

### Notes:

*Unit testing is essential for maintaining code quality and catching regressions early in development.*

## 10.2 MVVM Architecture

Model-View-ViewModel (MVVM) architecture separates concerns and makes SwiftUI apps more testable and maintainable.

### Code Example:

```
import SwiftUI
import Combine

// MARK: - Model
struct User: Identifiable, Codable, Equatable {
    let id: UUID
    var name: String
    var email: String
    var avatar: URL?
    var isActive: Bool

    init(name: String, email: String, avatar: URL? = nil, isActive: Bool = true) {
        self.id = UUID()
        self.name = name
        self.email = email
        self.avatar = avatar
        self.isActive = isActive
    }
}
```

```
// MARK: - Service Layer
protocol UserServiceProtocol {
    func fetchUsers() async throws -> [User]
    func createUser(_ user: User) async throws -> User
    func updateUser(_ user: User) async throws -> User
    func deleteUser(id: UUID) async throws
}

class UserService: UserServiceProtocol {
    func fetchUsers() async throws -> [User] {
        // Simulate network request
        try await Task.sleep(nanoseconds: 1_000_000_000)
        return [
            User(name: "John Doe", email: "john@example.com"),
            User(name: "Jane Smith", email: "jane@example.com"),
            User(name: "Bob Johnson", email: "bob@example.com")
        ]
    }

    func createUser(_ user: User) async throws -> User {
        // Simulate API call
        try await Task.sleep(nanoseconds: 500_000_000)
        return user
    }

    func updateUser(_ user: User) async throws -> User {
        try await Task.sleep(nanoseconds: 500_000_000)
        return user
    }

    func deleteUser(id: UUID) async throws {
        try await Task.sleep(nanoseconds: 500_000_000)
    }
}

// MARK: - ViewModel
class UserListViewModel: ObservableObject {
    @Published var users: [User] = []
    @Published var isLoading = false
    @Published var errorMessage: String?
    @Published var searchText = ""

    private let userService: UserServiceProtocol
    private var cancellables = Set<AnyCancellable>()

    // Computed properties
    var filteredUsers: [User] {
        if searchText.isEmpty {
            return users
        }
        return users.filter { user in
            user.name.localizedCaseInsensitiveContains(searchText) ||
            user.email.localizedCaseInsensitiveContains(searchText)
        }
    }

    var activeUsersCount: Int {
        users.filter { $0.isActive }.count
    }

    init(userService: UserServiceProtocol = UserService()) {
        self.userService = userService
        setupSearchDebouncing()
    }

    private func setupSearchDebouncing() {
        $searchText
            .debounce(for: .milliseconds(300), scheduler: RunLoop.main)
            .removeDuplicates()
            .sink { [weak self] _ in
                self?.objectWillChange.send()
            }
            .store(in: &cancellables)
    }

    // MARK: - User Actions
    @MainActor
    func loadUsers() async {
        isLoading = true
        errorMessage = nil

        do {
            users = try await userService.fetchUsers()
        } catch {
            errorMessage = "Failed to load users: \(error.localizedDescription)"
        }

        isLoading = false
    }

    @MainActor
    func addUser(name: String, email: String) async {
        let newUser = User(name: name, email: email)

        do {
            let createdUser = try await userService.createUser(newUser)
            users.append(createdUser)
        } catch {
            errorMessage = "Failed to add user: \(error.localizedDescription)"
        }
    }

    @MainActor
    func updateUser(_ user: User) async {
        do {
            let updatedUser = try await userService.updateUser(user)
            if let index = users.firstIndex(where: { $0.id == updatedUser.id }) {
                users[index] = updatedUser
            }
        } catch {
            errorMessage = "Failed to update user: \(error.localizedDescription)"
        }
    }
    @MainActor
    func deleteUser(_ user: User) async {
        do {
            try await userService.deleteUser(id: user.id)
            users.removeAll { $0.id == user.id }
        } catch {
            errorMessage = "Failed to delete user: \(error.localizedDescription)"
        }
    }

    func toggleUserActive(_ user: User) {
        if let index = users.firstIndex(where: { $0.id == user.id }) {
            users[index].isActive.toggle()
        }
    }

    func clearError() {
        errorMessage = nil
    }
}

// MARK: - View
struct UserListView: View {
    @StateObject private var viewModel = UserListViewModel()
    @State private var showingAddUser = false

    var body: some View {
        NavigationView {
            VStack {
                SearchBar(text: $viewModel.searchText)

                UserStatsView(
                    totalUsers: viewModel.users.count,
                    activeUsers: viewModel.activeUsersCount
                )

                if viewModel.isLoading {
                    ProgressView("Loading users...")
                        .frame(maxWidth: .infinity, maxHeight: .infinity)
                } else {
                    UserList(
                        users: viewModel.filteredUsers,
                        onToggleActive: { user in
                            viewModel.toggleUserActive(user)
                        },
                        onDelete: { user in
                            Task {
                                await viewModel.deleteUser(user)
                            }
                        }
                    )
                }
            }
            .navigationTitle("Users")
            .toolbar {
                ToolbarItem(placement: .navigationBarTrailing) {
                    Button("Add") {
                        showingAddUser = true
                    }
                }
            }
            .sheet(isPresented: $showingAddUser) {
                AddUserView { name, email in
                    Task {
                        await viewModel.addUser(name: name, email: email)
                    }
                }
            }
            .alert("Error", isPresented: .constant(viewModel.errorMessage != nil)) {
                Button("OK") {
                    viewModel.clearError()
                }
            } message: {
                Text(viewModel.errorMessage ?? "")
            }
        }
        .task {
            await viewModel.loadUsers()
        }
    }
}

// MARK: - Supporting Views
struct UserStatsView: View {
    let totalUsers: Int
    let activeUsers: Int

    var body: some View {
        HStack {
            StatCard(title: "Total", value: totalUsers, color: .blue)
            StatCard(title: "Active", value: activeUsers, color: .green)
        }
        .padding()
    }
}

struct StatCard: View {
    let title: String
    let value: Int
    let color: Color

    var body: some View {
        VStack {
            Text("\(value)")
                .font(.largeTitle)
                .fontWeight(.bold)
                .foregroundColor(color)

            Text(title)
                .font(.caption)
                .foregroundColor(.secondary)
        }
        .frame(maxWidth: .infinity)
        .padding()
```

```
            .background(Color(.systemGray6))
            .cornerRadius(10)
        }
    }
```

## Key Points:

**• MVVM separates presentation logic from view code**
**• ViewModels handle business logic and state management**
**• ObservableObject enables reactive UI updates**
**• Dependency injection makes code more testable**
**• Service layer abstracts data access logic**

## Notes:

*MVVM architecture makes SwiftUI apps more maintainable, testable, and follows separation of concerns principles.*

# PART V: NETWORKING & APIs

# Chapter 8: Networking

## 8.1 URLSession

URLSession is the foundation of networking in iOS. It provides APIs for making HTTP requests with modern async/await support.

### Code Example:

```
import Foundation

// Basic URLSession with async/await
class NetworkManager {
    static let shared = NetworkManager()
    private init() {}

    // Simple GET request
    func fetchData(from url: URL) async throws -> Data {
        let (data, response) = try await URLSession.shared.data(from: url)

        guard let httpResponse = response as? HTTPURLResponse,
              httpResponse.statusCode == 200 else {
            throw NetworkError.invalidResponse
        }

        return data
    }

    // POST request with JSON
    func postJSON<T: Codable>(to url: URL, body: T) async throws -> Data {
        var request = URLRequest(url: url)
        request.httpMethod = "POST"
        request.setValue("application/json", forHTTPHeaderField: "Content-Type")
        request.setValue("Bearer \(AuthManager.token)", forHTTPHeaderField: "Authorization")

        request.httpBody = try JSONEncoder().encode(body)

        let (data, response) = try await URLSession.shared.data(for: request)

        guard let httpResponse = response as? HTTPURLResponse,
              200...299 ~= httpResponse.statusCode else {
            throw NetworkError.serverError(response)
        }

        return data
    }

    // Download file with progress
    func downloadFile(from url: URL) -> AsyncThrowingStream<DownloadProgress, Error> {
        AsyncThrowingStream { continuation in
            let task = URLSession.shared.downloadTask(with: url) { localURL, response, error
```

```
                if let error = error {
                    continuation.finish(throwing: error)
                    return
                }

                guard let localURL = localURL else {
                    continuation.finish(throwing: NetworkError.noData)
                    return
                }

                // Move file to permanent location
                // continuation.yield(.completed(localURL))
                continuation.finish()
            }

            task.resume()
        }
    }

    // URLSession with custom configuration
    func createCustomSession() -> URLSession {
        let config = URLSessionConfiguration.default
        config.timeoutIntervalForRequest = 30
        config.timeoutIntervalForResource = 60
        config.httpMaximumConnectionsPerHost = 5
        config.requestCachePolicy = .reloadIgnoringLocalCacheData

        return URLSession(configuration: config)
    }

    // Retry mechanism
    func fetchWithRetry<T: Codable>(url: URL, type: T.Type, maxRetries: Int = 3) async throw
        var lastError: Error?

        for attempt in 1...maxRetries {
            do {
                let data = try await fetchData(from: url)
                return try JSONDecoder().decode(T.self, from: data)
            } catch {
                lastError = error
                if attempt < maxRetries {
                    let delay = Double(attempt * 2) // Exponential backoff
                    try await Task.sleep(nanoseconds: UInt64(delay * 1_000_000_000))
                }
            }
        }

        throw lastError ?? NetworkError.maxRetriesExceeded
    }
}

// Error handling
enum NetworkError: Error, LocalizedError {
    case invalidURL
    case noData
    case invalidResponse
    case serverError(URLResponse?)
```

```
        case decodingError
        case maxRetriesExceeded

        var errorDescription: String? {
            switch self {
            case .invalidURL:
                return "Invalid URL"
            case .noData:
                return "No data received"
            case .invalidResponse:
                return "Invalid response"
            case .serverError:
                return "Server error"
            case .decodingError:
                return "Failed to decode data"
            case .maxRetriesExceeded:
                return "Maximum retry attempts exceeded"
            }
        }
    }

    // Progress tracking
    struct DownloadProgress {
        let bytesWritten: Int64
        let totalBytesWritten: Int64
        let totalBytesExpectedToWrite: Int64

        var progress: Double {
            guard totalBytesExpectedToWrite > 0 else { return 0 }
            return Double(totalBytesWritten) / Double(totalBytesExpectedToWrite)
        }
    }
```

## Key Points:

- **async/await makes networking code more readable and maintainable**
- **Always handle HTTP status codes and potential errors**
- **URLSession configuration allows customization of timeouts and caching**
- **Implement retry mechanisms for robust networking**

## Notes:

*Modern Swift networking leverages async/await for cleaner asynchronous code without callback hell.*

# APPENDIX: DATA STRUCTURES & ALGORITHMS (100+ PROBLEMS)

This comprehensive appendix contains 100+ carefully selected Data Structures and Algorithms problems with complete Swift solutions. Each problem includes detailed problem statement, optimized Swift implementation, complexity analysis, and algorithmic explanation. The problems are organized by topic and difficulty to facilitate systematic learning and technical interview preparation.

## A.1 Array Problems (20 Problems)

### A.1.1 Two Sum ■

Given an array of integers nums and an integer target, return indices of the two numbers that add up to target.

#### Code Example:

```
func twoSum(_ nums: [Int], _ target: Int) -> [Int] {
    var hashMap: [Int: Int] = [:]

    for (index, num) in nums.enumerated() {
        let complement = target - num
        if let complementIndex = hashMap[complement] {
            return [complementIndex, index]
        }
        hashMap[num] = index
    }
    return []
}

// Test case
let nums = [2, 7, 11, 15], target = 9
print(twoSum(nums, target)) // [0, 1]
```

#### Key Points:

**• Time: O(n)**
**• Space: O(n)**
**• Hash table approach**

#### Notes:

*Use hash map to store numbers and their indices for O(1) complement lookup.*

### A.1.2 Best Time to Buy and Sell Stock ■

Find the maximum profit from buying and selling stock once.

#### Code Example:

```
func maxProfit(_ prices: [Int]) -> Int {
    guard !prices.isEmpty else { return 0 }
    var minPrice = prices[0]
    var maxProfit = 0
    for price in prices {
        if price < minPrice {
            minPrice = price
        } else if price - minPrice > maxProfit {
            maxProfit = price - minPrice
        }
    }
    return maxProfit
}

// Test case
let prices = [7, 1, 5, 3, 6, 4]
print(maxProfit(prices)) // 5
```

### Key Points:

- **Time: O(n)**
- **Space: O(1)**
- **Single pass solution**

### Notes:

*Track minimum price and calculate maximum profit at each step.*

## A.1.3 Contains Duplicate ■

Return true if any value appears at least twice in the array.

### Code Example:

```
func containsDuplicate(_ nums: [Int]) -> Bool {
    return Set(nums).count != nums.count
}

// Alternative approach with early exit
func containsDuplicateOptimal(_ nums: [Int]) -> Bool {
    var seen: Set<Int> = []
    for num in nums {
        if seen.contains(num) { return true }
        seen.insert(num)
    }
    return false
}
```

### Key Points:

- **Time: O(n)**
- **Space: O(n)**
- **Set for duplicate detection**

### Notes:

*Use Set to detect duplicates efficiently - either compare counts or track seen elements.*

## A.1.4 Product of Array Except Self ■■

Return array where each element is the product of all other elements.

### Code Example:

```
func productExceptSelf(_ nums: [Int]) -> [Int] {
    let n = nums.count
    var result = Array(repeating: 1, count: n)

    // Left products
    for i in 1..<n {
        result[i] = result[i-1] * nums[i-1]
    }

    // Right products
    var right = 1
    for i in stride(from: n-1, through: 0, by: -1) {
        result[i] *= right
        right *= nums[i]
    }

    return result
}
```

### Key Points:

**• Time: O(n)**
**• Space: O(1)**
**• Two-pass solution**

### Notes:

*Calculate left products, then multiply by right products in reverse pass.*

## A.1.5 Maximum Subarray (Kadane's) ■■

Find the contiguous subarray with the largest sum.

### Code Example:

```
func maxSubArray(_ nums: [Int]) -> Int {
    var maxSum = nums[0]
    var currentSum = nums[0]

    for i in 1..<nums.count {
        currentSum = max(nums[i], currentSum + nums[i])
        maxSum = max(maxSum, currentSum)
    }
    return maxSum
}

// Return actual subarray
func maxSubArrayWithIndices(_ nums: [Int]) -> [Int] {
    var maxSum = nums[0]
    var currentSum = nums[0]
    var start = 0, end = 0, tempStart = 0

    for i in 1..<nums.count {
        if nums[i] > currentSum + nums[i] {
            currentSum = nums[i]
            tempStart = i
        } else {
            currentSum += nums[i]
        }

        if currentSum > maxSum {
            maxSum = currentSum
            start = tempStart
            end = i
        }
    }

    return Array(nums[start...end])
}
```

### Key Points:

- **Time: O(n)**
- **Space: O(1)**
- **Kadane's algorithm**

### Notes:

*At each position, decide whether to extend current subarray or start new one.*

## A.1.6 Maximum Product Subarray ■■

Find the contiguous subarray with the largest product.

### Code Example:

```
func maxProduct(_ nums: [Int]) -> Int {
    var maxProd = nums[0]
    var minProd = nums[0]
    var result = nums[0]

    for i in 1..<nums.count {
        let temp = maxProd
```

```
            maxProd = max(nums[i], max(maxProd * nums[i], minProd * nums[i]))
            minProd = min(nums[i], min(temp * nums[i], minProd * nums[i]))
            result = max(result, maxProd)
        }
        return result
    }

    // Test case
    let nums = [2, 3, -2, 4]
    print(maxProduct(nums)) // 6
```

### Key Points:

- **Time: O(n)**
- **Space: O(1)**
- **Track both max and min**

### Notes:

*Track both maximum and minimum products due to negative numbers.*

## A.1.7 Find Min in Rotated Sorted Array ■■

Find the minimum element in a rotated sorted array.

### Code Example:

```
func findMin(_ nums: [Int]) -> Int {
    var left = 0
    var right = nums.count - 1

    while left < right {
        let mid = left + (right - left) / 2

        if nums[mid] > nums[right] {
            left = mid + 1
        } else {
            right = mid
        }
    }

    return nums[left]
}

// Test case
let nums = [3, 4, 5, 1, 2]
print(findMin(nums)) // 1
```

### Key Points:

- **Time: O(log n)**
- **Space: O(1)**
- **Binary search**

## Notes:

*Use binary search to find the inflection point where rotation occurred.*

## A.1.8 Search in Rotated Sorted Array ■■

Search for a target in a rotated sorted array.

### Code Example:

```
func search(_ nums: [Int], _ target: Int) -> Int {
    var left = 0
    var right = nums.count - 1

    while left <= right {
        let mid = left + (right - left) / 2

        if nums[mid] == target { return mid }

        if nums[left] <= nums[mid] {
            if nums[left] <= target && target < nums[mid] {
                right = mid - 1
            } else {
                left = mid + 1
            }
        } else {
            if nums[mid] < target && target <= nums[right] {
                left = mid + 1
            } else {
                right = mid - 1
            }
        }
    }

    return -1
}

// Test case
let nums = [4, 5, 6, 7, 0, 1, 2], target = 0
print(search(nums, target)) // 4
```

### Key Points:

- **Time: O(log n)**
- **Space: O(1)**
- **Modified binary search**

### Notes:

*Determine which half is sorted, then decide which half to search.*

## A.1.9 3Sum ■■

Find all unique triplets that sum to zero.

### Code Example:

```
func threeSum(_ nums: [Int]) -> [[Int]] {
    let sorted = nums.sorted()
    var result: [[Int]] = []

    for i in 0..<sorted.count - 2 {
        if i > 0 && sorted[i] == sorted[i-1] { continue }

        var left = i + 1
        var right = sorted.count - 1

        while left < right {
            let sum = sorted[i] + sorted[left] + sorted[right]

            if sum == 0 {
                result.append([sorted[i], sorted[left], sorted[right]])

                while left < right && sorted[left] == sorted[left + 1] {
                    left += 1
                }
                while left < right && sorted[right] == sorted[right - 1] {
                    right -= 1
                }

                left += 1
                right -= 1
            } else if sum < 0 {
                left += 1
            } else {
                right -= 1
            }
        }
    }

    return result
}

// Test case
let nums = [-1, 0, 1, 2, -1, -4]
print(threeSum(nums)) // [[-1, -1, 2], [-1, 0, 1]]
```

### Key Points:

**• Time: O($n^2$)**
**• Space: O(1)**
**• Two pointers technique**

### Notes:

*Sort array, then use two pointers for each fixed element to find triplets.*

## A.1.10 Container With Most Water ■■

Find two lines that form a container holding the most water.

### Code Example:

```
func maxArea(_ height: [Int]) -> Int {
    var left = 0
    var right = height.count - 1
    var maxWater = 0

    while left < right {
        let water = min(height[left], height[right]) * (right - left)
        maxWater = max(maxWater, water)

        if height[left] < height[right] {
            left += 1
        } else {
            right -= 1
        }
    }

    return maxWater
}

// Test case
let height = [1, 8, 6, 2, 5, 4, 8, 3, 7]
print(maxArea(height)) // 49
```

### Key Points:

**• Time: O(n)**
**• Space: O(1)**
**• Two pointers approach**

### Notes:

*Use two pointers moving inward, always moving the pointer with smaller height.*

## A.1.11 Trapping Rain Water ■■■

Calculate how much water can be trapped after it rains.

### Code Example:

```
func trap(_ height: [Int]) -> Int {
    guard height.count > 2 else { return 0 }

    var left = 0, right = height.count - 1
    var leftMax = 0, rightMax = 0
    var water = 0

    while left < right {
        if height[left] < height[right] {
            if height[left] >= leftMax {
                leftMax = height[left]
            } else {
                water += leftMax - height[left]
            }
            left += 1
        } else {
            if height[right] >= rightMax {
                rightMax = height[right]
            } else {
                water += rightMax - height[right]
            }
            right -= 1
        }
    }

    return water
}
```

### Key Points:

**• Time: O(n)**
**• Space: O(1)**
**• Two pointers with max tracking**

### Notes:

*Use two pointers tracking maximum heights from both ends.*

## A.1.12 Merge Intervals ■■

Merge overlapping intervals.

### Code Example:

```
func merge(_ intervals: [[Int]]) -> [[Int]] {
    guard !intervals.isEmpty else { return [] }

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

## Key Points:

- **Time: O(n log n)**
- **Space: O(1)**
- **Sort and merge**

## Notes:

*Sort intervals by start time, then merge overlapping ones.*

## A.1.13 Insert Interval ■■

Insert a new interval into sorted non-overlapping intervals.

## Code Example:

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

    // Add all intervals before the new interval
    while i < intervals.count && intervals[i][1] < current[0] {
        result.append(intervals[i])
        i += 1
    }

    // Merge overlapping intervals
    while i < intervals.count && intervals[i][0] <= current[1] {
        current[0] = min(current[0], intervals[i][0])
        current[1] = max(current[1], intervals[i][1])
        i += 1
    }
    result.append(current)

    // Add remaining intervals
    while i < intervals.count {
        result.append(intervals[i])
        i += 1
    }

    return result
}
```

### Key Points:

- **Time: O(n)**
- **Space: O(1)**
- **Three-phase insertion**

### Notes:

*Insert and merge by finding correct position and handling overlaps.*

## A.1.14 Rotate Array ■

Rotate array to the right by k steps in-place.

### Code Example:

```
func rotate(_ nums: inout [Int], _ k: Int) {
    let n = nums.count
    let k = k % n

    func reverse(_ start: Int, _ end: Int) {
        var left = start, right = end
        while left < right {
            nums.swapAt(left, right)
            left += 1
            right -= 1
        }
    }

    // Reverse entire array
    reverse(0, n - 1)
    // Reverse first k elements
    reverse(0, k - 1)
    // Reverse remaining elements
    reverse(k, n - 1)
}

// Test case
var nums = [1, 2, 3, 4, 5, 6, 7]
rotate(&nums, 3)
print(nums) // [5, 6, 7, 1, 2, 3, 4]
```

### Key Points:

- **Time: O(n)**
- **Space: O(1)**
- **Three reversals technique**

### Notes:

*Use three reversals: entire array, first k, last n-k elements.*

## A.1.15 Jump Game ■■

Determine if you can reach the last index.

### Code Example:

```
func canJump(_ nums: [Int]) -> Bool {
    var maxReach = 0

    for i in 0..<nums.count {
        if i > maxReach { return false }
        maxReach = max(maxReach, i + nums[i])
        if maxReach >= nums.count - 1 { return true }
    }

    return maxReach >= nums.count - 1
}

// Test case
let nums1 = [2, 3, 1, 1, 4] // true
let nums2 = [3, 2, 1, 0, 4] // false
print(canJump(nums1), canJump(nums2))
```

### Key Points:

- **Time: O(n)**
- **Space: O(1)**
- **Greedy approach**

### Notes:

*Track maximum reachable index, return false if gap found.*

## A.1.16 Spiral Matrix ■■

Return all elements of matrix in spiral order.

### Code Example:

```
func spiralOrder(_ matrix: [[Int]]) -> [Int] {
    guard !matrix.isEmpty else { return [] }

    var result: [Int] = []
    var top = 0, bottom = matrix.count - 1
    var left = 0, right = matrix[0].count - 1

    while top <= bottom && left <= right {
        // Right
        for col in left...right {
            result.append(matrix[top][col])
        }
        top += 1

        // Down
        for row in top...bottom {
            result.append(matrix[row][right])
        }
        right -= 1

        // Left (if still valid row)
        if top <= bottom {
            for col in stride(from: right, through: left, by: -1) {
                result.append(matrix[bottom][col])
            }
            bottom -= 1
        }

        // Up (if still valid column)
        if left <= right {
            for row in stride(from: bottom, through: top, by: -1) {
                result.append(matrix[row][left])
            }
            left += 1
        }
    }

    return result
}
```

### Key Points:

**• Time: O(m×n)**
**• Space: O(1)**
**• Four-direction traversal**

### Notes:

*Use four boundaries and traverse in spiral pattern.*

## A.1.17 Meeting Rooms ■

Determine if person can attend all meetings.

### Code Example:

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

    let sorted = intervals.sorted { $0[0] < $1[0] }
    for i in 1..<sorted.count {
        if sorted[i][0] < sorted[i-1][1] {
            return false
        }
    }

    return true
}

// Test case
let meetings1 = [[0,30],[5,10],[15,20]] // false
let meetings2 = [[7,10],[2,4]] // true
print(canAttendMeetings(meetings1))
```

### Key Points:

**• Time: O(n log n)**
**• Space: O(1)**
**• Sort by start time**

### Notes:

*Sort meetings by start time, check for overlaps.*

## A.1.18 Meeting Rooms II ■■

Find minimum number of meeting rooms needed.

### Code Example:

```
func minMeetingRooms(_ intervals: [[Int]]) -> Int {
    var starts = intervals.map { $0[0] }.sorted()
    var ends = intervals.map { $0[1] }.sorted()

    var rooms = 0
    var endPtr = 0

    for start in starts {
        if start >= ends[endPtr] {
            endPtr += 1
        } else {
            rooms += 1
        }
    }

    return rooms
}

// Test case
let meetings = [[0,30],[5,10],[15,20]]
print(minMeetingRooms(meetings)) // 2
```

### Key Points:

- **Time: O(n log n)**
- **Space: O(n)**
- **Two pointers on sorted arrays**

### Notes:

*Separate start/end times, use two pointers to track concurrent meetings.*

## A.1.19 Non-overlapping Intervals ■■

Minimum number of intervals to remove to make non-overlapping.

### Code Example:

```
func eraseOverlapIntervals(_ intervals: [[Int]]) -> Int {
    guard intervals.count > 1 else { return 0 }

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

    for i in 1..<sorted.count {
        if sorted[i][0] < end {
            count += 1
        } else {
            end = sorted[i][1]
        }
    }

    return count
}

// Test case
let intervals = [[1,2],[2,3],[3,4],[1,3]]
print(eraseOverlapIntervals(intervals)) // 1
```

### Key Points:

- **Time: O(n log n)**
- **Space: O(1)**
- **Greedy by end time**

### Notes:

*Sort by end time, greedily keep intervals that end earliest.*

## A.1.20 Jump Game II ■■

Find minimum number of jumps to reach the last index.

## Code Example:

```
func jump(_ nums: [Int]) -> Int {
    var jumps = 0
    var currentEnd = 0
    var farthest = 0

    for i in 0..<nums.count - 1 {
        farthest = max(farthest, i + nums[i])

        if i == currentEnd {
            jumps += 1
            currentEnd = farthest
        }
    }

    return jumps
}

// Test case
let nums1 = [2, 3, 1, 1, 4] // 2
let nums2 = [2, 3, 0, 1, 4] // 2
print(jump(nums1), jump(nums2))
```

## Key Points:

**• Time: O(n)**
**• Space: O(1)**
**• Greedy BFS approach**

## Notes:

*Track current jump range and farthest reachable, increment jumps at range end.*

## A.2 String Problems (15 Problems)

### A.2.1 Valid Anagram ■

Determine if two strings are anagrams of each other.

#### Code Example:

```
func isAnagram(_ s: String, _ t: String) -> Bool {
    guard s.count == t.count else { return false }

    var charCount: [Character: Int] = [:]

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

    for char in t {
        charCount[char, default: 0] -= 1
        if charCount[char]! < 0 { return false }
    }

    return charCount.values.allSatisfy { $0 == 0 }
}

// Test case
print(isAnagram("anagram", "nagaram")) // true
print(isAnagram("rat", "car")) // false
```

#### Key Points:

**• Time: O(n)**
**• Space: O(1)**
**• Character frequency counting**

#### Notes:

*Count character frequencies and verify they match exactly.*

### A.2.2 Valid Palindrome ■

Check if string is palindrome ignoring non-alphanumeric characters.

#### Code Example:

```
func isPalindrome(_ s: String) -> Bool {
    let cleaned = s.lowercased().filter { $0.isAlphanumeric }
```

```
        return cleaned == String(cleaned.reversed())
    }

    // Alternative two-pointer approach
    func isPalindromeOptimal(_ s: String) -> Bool {
        let chars = Array(s.lowercased())
        var left = 0, right = chars.count - 1

        while left < right {
            while left < right && !chars[left].isAlphanumeric {
                left += 1
            }
            while left < right && !chars[right].isAlphanumeric {
                right -= 1
            }

            if chars[left] != chars[right] { return false }
            left += 1
            right -= 1
        }

        return true
    }
```

### Key Points:

- **Time: O(n)**
- **Space: O(1)**
- **Two pointers technique**

### Notes:

*Use two pointers moving inward, skipping non-alphanumeric characters.*

## A.2.3 Longest Substring Without Repeating ■■

Find length of longest substring without repeating characters.

### Code Example:

```
    func lengthOfLongestSubstring(_ s: String) -> Int {
        var charIndex: [Character: Int] = [:]
        var maxLen = 0
        var start = 0

        for (i, char) in s.enumerated() {
            if let lastIndex = charIndex[char], lastIndex >= start {
                start = lastIndex + 1
            }
            charIndex[char] = i
            maxLen = max(maxLen, i - start + 1)
        }
        return maxLen
    }

    // Test cases
    print(lengthOfLongestSubstring("abcabcbb")) // 3
    print(lengthOfLongestSubstring("bbbbb")) // 1
    print(lengthOfLongestSubstring("pwwkew")) // 3
```

### Key Points:

- **Time: O(n)**
- **Space: O(min(m,n))**
- **Sliding window with HashMap**

### Notes:

*Use sliding window with character index tracking to avoid duplicates.*

## A.2.4 Longest Repeating Character Replacement ■■

Find longest substring with at most k character replacements.

### Code Example:

```
func characterReplacement(_ s: String, _ k: Int) -> Int {
    var charCount: [Character: Int] = [:]
    var maxCount = 0
    var left = 0
    var maxLen = 0
    let chars = Array(s)

    for right in 0..<chars.count {
        charCount[chars[right], default: 0] += 1
        maxCount = max(maxCount, charCount[chars[right]]!)

        if right - left + 1 - maxCount > k {
            charCount[chars[left]]! -= 1
            left += 1
        }

        maxLen = max(maxLen, right - left + 1)
    }

    return maxLen
}

// Test case
print(characterReplacement("ABAB", 2)) // 4
print(characterReplacement("AABABBA", 1)) // 4
```

### Key Points:

- **Time: O(n)**

- **Space: O(1)**
- **Sliding window optimization**

### Notes:

*Expand window while valid, shrink when replacements exceed k.*

## A.2.5 Minimum Window Substring ■■■

Find minimum window in s that contains all characters of t.

### Code Example:

```
func minWindow(_ s: String, _ t: String) -> String {
    guard s.count >= t.count else { return "" }

    var tCount: [Character: Int] = [:]
    for char in t {
        tCount[char, default: 0] += 1
    }

    var required = tCount.count
    var left = 0, right = 0
    var formed = 0
    var windowCounts: [Character: Int] = [:]

    var ans: (Int, Int, Int) = (-1, 0, 0) // length, left, right
    let sArray = Array(s)

    while right < sArray.count {
        let char = sArray[right]
        windowCounts[char, default: 0] += 1

        if let count = tCount[char], windowCounts[char] == count {
            formed += 1
        }

        while left <= right && formed == required {
            if ans.0 == -1 || right - left + 1 < ans.0 {
                ans = (right - left + 1, left, right)
            }

            let leftChar = sArray[left]
            windowCounts[leftChar]! -= 1
            if let count = tCount[leftChar], windowCounts[leftChar]! < count {
                formed -= 1
            }
            left += 1
        }
        right += 1
    }

    return ans.0 == -1 ? "" : String(sArray[ans.1...ans.2])
}
```

### Key Points:

**• Time: O(|s| + |t|)**
**• Space: O(|s| + |t|)**
**• Sliding window technique**

### Notes:

*Use sliding window to find minimum valid window containing all characters.*

## A.2.6 Group Anagrams ■■

Group strings that are anagrams of each other.

### Code Example:

```
func groupAnagrams(_ strs: [String]) -> [[String]] {
    var anagramMap: [String: [String]] = [:]

    for str in strs {
        let sortedStr = String(str.sorted())
        anagramMap[sortedStr, default: []].append(str)
    }

    return Array(anagramMap.values)
}

// Alternative: Using character frequency as key
func groupAnagramsFreq(_ strs: [String]) -> [[String]] {
    var groups: [[Int]: [String]] = [:]

    for str in strs {
        var count = Array(repeating: 0, count: 26)
        for char in str {
            count[Int(char.asciiValue! - 97)] += 1
        }
        groups[count, default: []].append(str)
    }

    return Array(groups.values)
}
```

### Key Points:

**• Time: O(n*k*log(k))**
**• Space: O(n*k)**
**• Sorting or frequency counting**

### Notes:

*Group strings by their sorted characters or frequency signature.*

## A.2.7 Valid Parentheses ■

Determine if parentheses are valid and properly nested.

### Code Example:

```
func isValid(_ s: String) -> Bool {
    var stack: [Character] = []
    let pairs: [Character: Character] = [")": "(", "}": "{", "]": "["]

    for char in s {
        if let openBracket = pairs[char] {
            if stack.isEmpty || stack.removeLast() != openBracket {
                return false
            }
        } else {
            stack.append(char)
        }
    }

    return stack.isEmpty
}

// Test cases
print(isValid("()")) // true
print(isValid("()[]{}")) // true
print(isValid("(]")) // false
```

### Key Points:

**• Time: O(n)**
**• Space: O(n)**
**• Stack data structure**

### Notes:

*Use stack to track opening brackets and match with closing ones.*

## A.2.8 Longest Palindromic Substring ■■

Find the longest palindromic substring.

### Code Example:

```
func longestPalindrome(_ s: String) -> String {
    guard s.count > 1 else { return s }
    let chars = Array(s)
```

```
        var start = 0, maxLen = 1

        func expandAroundCenter(_ left: Int, _ right: Int) -> Int {
            var l = left, r = right
            while l >= 0 && r < chars.count && chars[l] == chars[r] {
                l -= 1
                r += 1
            }
            return r - l - 1
        }

        for i in 0..<chars.count {
            let len1 = expandAroundCenter(i, i)     // odd length
            let len2 = expandAroundCenter(i, i + 1) // even length
            let len = max(len1, len2)

            if len > maxLen {
                maxLen = len
                start = i - (len - 1) / 2
            }
        }

        return String(chars[start..<start + maxLen])
    }

    // Test case
    print(longestPalindrome("babad")) // "bab" or "aba"
```

### Key Points:

**• Time: O(n²)**
**• Space: O(1)**
**• Expand around centers**

### Notes:

*Check all possible centers and expand outward to find longest palindrome.*

## A.2.9 Palindromic Substrings ■■

Count the number of palindromic substrings.

### Code Example:

```
    func countSubstrings(_ s: String) -> Int {
        let chars = Array(s)
        var count = 0

        func expandAroundCenter(_ left: Int, _ right: Int) {
            var l = left, r = right
            while l >= 0 && r < chars.count && chars[l] == chars[r] {
                count += 1
                l -= 1
                r += 1
            }
        }

        for i in 0..<chars.count {
            expandAroundCenter(i, i)     // odd length palindromes
            expandAroundCenter(i, i + 1) // even length palindromes
        }

        return count
    }

    // Test case
    print(countSubstrings("abc")) // 3
    print(countSubstrings("aaa")) // 6
```

### Key Points:

- **Time: O(n²)**
- **Space: O(1)**
- **Expand around centers**

### Notes:

*Expand around all possible centers counting palindromes.*

## A.2.10 Encode and Decode Strings ■■

Design an algorithm to encode/decode list of strings.

### Code Example:

```
class Codec {
    func encode(_ strs: [String]) -> String {
        var encoded = ""
        for str in strs {
            encoded += "\(str.count)#\(str)"
        }
        return encoded
    }

    func decode(_ s: String) -> [String] {
        var result: [String] = []
        var i = 0
        let chars = Array(s)

        while i < chars.count {
            var j = i
            while chars[j] != "#" {
                j += 1
            }
            let len = Int(String(chars[i..<j]))!
            result.append(String(chars[j+1..<j+1+len]))
            i = j + 1 + len
        }

        return result
    }
}

// Test case
let codec = Codec()
let encoded = codec.encode(["hello", "world"])
print(codec.decode(encoded)) // ["hello", "world"]
```

### Key Points:

- **Time: O(n)**
- **Space: O(1)**
- **Length-prefixed encoding**

### Notes:

*Use length-prefixed format: length#string for safe encoding.*

## A.2.11 String to Integer (atoi) ■■

Convert string to 32-bit signed integer.

### Code Example:

```
func myAtoi(_ s: String) -> Int {
    let trimmed = s.trimmingCharacters(in: .whitespaces)
    guard !trimmed.isEmpty else { return 0 }

    var result = 0
    var isNegative = false
    var startIndex = trimmed.startIndex

    // Check sign
    if trimmed.first == "+" || trimmed.first == "-" {
        isNegative = trimmed.first == "-"
        startIndex = trimmed.index(after: startIndex)
    }

    let intMax = Int32.max
    let intMin = Int32.min

    for char in trimmed[startIndex...] {
        guard char.isNumber else { break }

        let digit = Int(String(char))!

        // Check overflow
        if result > Int(intMax / 10) ||
           (result == Int(intMax / 10) && digit > Int(intMax % 10)) {
            return isNegative ? Int(intMin) : Int(intMax)
        }

        result = result * 10 + digit
    }

    return isNegative ? -result : result
}

// Test cases
print(myAtoi("42")) // 42
print(myAtoi("-42")) // -42
print(myAtoi("4193 with words")) // 4193
```

### Key Points:

- **Time: O(n)**
- **Space: O(1)**
- **Character parsing with overflow check**

### Notes:

*Parse characters while handling signs, whitespace, and overflow.*

## A.2.12 Reverse Words in String ■■

Reverse the order of words in a string.

### Code Example:

```
func reverseWords(_ s: String) -> String {
    let words = s.split(separator: " ")
    return words.reversed().joined(separator: " ")
}

// In-place approach (conceptual - Swift strings are immutable)
func reverseWordsOptimal(_ s: String) -> String {
    // Trim and split by multiple spaces
    let trimmed = s.trimmingCharacters(in: .whitespaces)
    let words = trimmed.components(separatedBy: .whitespaces)
                      .filter { !$0.isEmpty }

    return words.reversed().joined(separator: " ")
}

// Test case
let input = "  hello world  "
print(reverseWords(input)) // "world hello"
```

### Key Points:

- **Time: O(n)**
- **Space: O(n)**

• **Split and reverse**

### Notes:

*Split by spaces, filter empty strings, reverse, and rejoin.*

## A.2.13 Implement strStr() ■

Find the first occurrence of needle in haystack.

### Code Example:

```
func strStr(_ haystack: String, _ needle: String) -> Int {
    guard !needle.isEmpty else { return 0 }
    guard haystack.count >= needle.count else { return -1 }

    let hayArray = Array(haystack)
    let needleArray = Array(needle)

    for i in 0...hayArray.count - needleArray.count {
        var match = true
        for j in 0..<needleArray.count {
            if hayArray[i + j] != needleArray[j] {
                match = false
                break
            }
        }
        if match { return i }
    }

    return -1
}

// KMP approach for optimal performance
func strStrKMP(_ haystack: String, _ needle: String) -> Int {
    // KMP implementation would go here
    return haystack.range(of: needle)?.lowerBound.utf16Offset(in: haystack) ?? -1
}

// Test case
print(strStr("hello", "ll")) // 2
print(strStr("aaaaa", "bba")) // -1
```

### Key Points:

• **Time: O(n*m) naive, O(n+m) KMP**
• **Space: O(1)**
• **String matching**

### Notes:

*Slide needle over haystack checking for matches at each position.*

## A.2.14 Text Justification ■■■

Format text to given width with full justification.

### Code Example:

```swift
func fullJustify(_ words: [String], _ maxWidth: Int) -> [String] {
    var result: [String] = []
    var i = 0

    while i < words.count {
        var currentLine: [String] = [words[i]]
        var currentLength = words[i].count
        i += 1

        // Pack as many words as possible
        while i < words.count &&
              currentLength + 1 + words[i].count <= maxWidth {
            currentLine.append(words[i])
            currentLength += 1 + words[i].count
            i += 1
        }

        // Justify the line
        if i == words.count || currentLine.count == 1 {
            // Last line or single word - left justify
            let line = currentLine.joined(separator: " ")
            let padding = String(repeating: " ", count: maxWidth - line.count)
            result.append(line + padding)
        } else {
            // Middle lines - full justify
            let totalSpaces = maxWidth - currentLine.reduce(0) { $0 + $1.count }
            let gaps = currentLine.count - 1
            let spacesPerGap = totalSpaces / gaps
            let extraSpaces = totalSpaces % gaps

            var line = ""
            for j in 0..<currentLine.count {
                line += currentLine[j]
                if j < gaps {
                    line += String(repeating: " ", count: spacesPerGap)
                    if j < extraSpaces {
                        line += " "
                    }
                }
            }
            result.append(line)
        }
    }

    return result
}
```

### Key Points:

- **Time: O(n)**
- **Space: O(1)**
- **Greedy packing with justification**

### Notes:

*Pack words into lines, then distribute spaces evenly for justification.*

## A.2.15 Regular Expression Matching ■■■

Implement regular expression matching with '.' and '*'.

### Code Example:

```
func isMatch(_ s: String, _ p: String) -> Bool {
    let sArray = Array(s)
    let pArray = Array(p)
    var memo: [[Int]] = Array(repeating: Array(repeating: -1, count: pArray.count + 1),
                              count: sArray.count + 1)

    func dp(_ i: Int, _ j: Int) -> Bool {
        if memo[i][j] != -1 {
            return memo[i][j] == 1
        }

        var result: Bool

        if j == pArray.count {
            result = i == sArray.count
        } else {
            let firstMatch = i < sArray.count &&
                             (pArray[j] == "." || sArray[i] == pArray[j])

            if j + 1 < pArray.count && pArray[j + 1] == "*" {
                result = dp(i, j + 2) || (firstMatch && dp(i + 1, j))
            } else {
                result = firstMatch && dp(i + 1, j + 1)
            }
        }

        memo[i][j] = result ? 1 : 0
        return result
    }

    return dp(0, 0)
}

// Test cases
print(isMatch("aa", "a")) // false
print(isMatch("aa", "a*")) // true
print(isMatch("ab", ".*")) // true
```

## Key Points:

- **Time: O(s*p)**
- **Space: O(s*p)**
- **Dynamic programming with memoization**

## Notes:

*Use recursion with memoization to handle . and * pattern matching.*

# A.3 Linked List Problems (12 Problems)

```
// ListNode Definition for Linked List Problems class ListNode { var val: Int var
next: ListNode? init() { self.val = 0; self.next = nil } init(_ val: Int) { self.val
= val; self.next = nil } init(_ val: Int, _ next: ListNode?) { self.val = val;
self.next = next } }
```

## A.3.1 Reverse Linked List ■

Reverse a singly linked list iteratively and recursively.

### Code Example:

```
// Iterative approach
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
}

// Recursive approach
func reverseListRecursive(_ head: ListNode?) -> ListNode? {
    guard let head = head, let next = head.next else {
        return head
    }

    let newHead = reverseListRecursive(next)
    next.next = head
    head.next = nil

    return newHead
}
```

### Key Points:

- **Time: O(n)**
- **Space: O(1) iterative, O(n) recursive**
- **Two pointers**

### Notes:

*Use three pointers to reverse links, or recursion with stack.*

## A.3.2 Merge Two Sorted Lists ■

Merge two sorted linked lists into one sorted list.

### Code Example:

```
func mergeTwoLists(_ list1: ListNode?, _ list2: ListNode?) -> ListNode? {
    let dummy = ListNode(0)
    var current = dummy
    var l1 = list1
    var l2 = list2

    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!
    }

    // Attach remaining nodes
    current.next = l1 ?? l2

    return dummy.next
}

// Recursive approach
func mergeTwoListsRecursive(_ l1: ListNode?, _ l2: ListNode?) -> ListNode? {
    guard let l1 = l1 else { return l2 }
    guard let l2 = l2 else { return l1 }

    if l1.val <= l2.val {
        l1.next = mergeTwoListsRecursive(l1.next, l2)
        return l1
    } else {
        l2.next = mergeTwoListsRecursive(l1, l2.next)
        return l2
    }
}
```

### Key Points:

**• Time: O(n + m)**
**• Space: O(1) iterative, O(n+m) recursive**
**• Two pointers**

### Notes:

*Compare values at each step and attach the smaller node.*

## A.3.3 Remove Nth Node From End ■■

Remove the nth node from the end in one pass.

### Code Example:

```
func removeNthFromEnd(_ head: ListNode?, _ n: Int) -> ListNode? {
    let dummy = ListNode(0)
    dummy.next = head
    var first: ListNode? = dummy
    var second: ListNode? = dummy

    // Move first pointer n+1 steps ahead
    for _ in 0...n {
        first = first?.next
    }

    // Move both pointers until first reaches end
    while first != nil {
        first = first?.next
        second = second?.next
    }

    // Remove the nth node
    second?.next = second?.next?.next

    return dummy.next
}
```

### Key Points:

**• Time: O(n)**
**• Space: O(1)**
**• Two pointers technique**

### Notes:

*Use two pointers with n+1 gap to find the node before target.*

## A.3.4 Linked List Cycle ■

Detect if linked list has a cycle using Floyd's algorithm.

### Code Example:

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

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

    return false
}

// Alternative using Set (less efficient)
func hasCycleSet(_ head: ListNode?) -> Bool {
    var visited = Set<ObjectIdentifier>()
    var current = head

    while let node = current {
        let id = ObjectIdentifier(node)
        if visited.contains(id) {
            return true
        }
        visited.insert(id)
        current = node.next
    }

    return false
}
```

### Key Points:

- **Time: O(n)**
- **Space: O(1)**
- **Floyd's tortoise and hare**

### Notes:

*Fast pointer moves twice as fast; they meet if cycle exists.*

## A.3.5 Linked List Cycle II ■■

Find the node where the cycle begins.

### Code Example:

```
func detectCycle(_ head: ListNode?) -> ListNode? {
    var slow = head
    var fast = head

    // Phase 1: Detect if cycle exists
    while fast != nil && fast?.next != nil {
        slow = slow?.next
        fast = fast?.next?.next

        if slow === fast {
            // Phase 2: Find cycle start
            var start = head
            while start !== slow {
                start = start?.next
                slow = slow?.next
            }
            return start
        }
    }

    return nil
}
```

### Key Points:

- **Time: O(n)**
- **Space: O(1)**
- **Floyd's algorithm extended**

### Notes:

*After detecting cycle, move one pointer to head and advance both one step.*

## A.3.6 Merge k Sorted Lists ■■■

Merge k sorted linked lists using divide and conquer.

### Code Example:

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

    var lists = lists.filter { $0 != nil }

    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.first ?? nil
}

// Helper function from problem A.3.2
func mergeTwoLists(_ l1: ListNode?, _ l2: ListNode?) -> ListNode? {
    let dummy = ListNode(0)
    var current = dummy
    var list1 = l1, list2 = l2
```

```
        while list1 != nil && list2 != nil {
            if list1!.val <= list2!.val {
                current.next = list1
                list1 = list1!.next
            } else {
                current.next = list2
                list2 = list2!.next
            }
            current = current.next!
        }

        current.next = list1 ?? list2
        return dummy.next
    }
```

### Key Points:

**• Time: O(n log k)**
**• Space: O(1)**
**• Divide and conquer**

### Notes:

*Merge lists pairwise until only one remains.*

## A.3.7 Remove Duplicates from Sorted List ■

Remove duplicates from a sorted linked list.

### Code Example:

```
    func deleteDuplicates(_ head: ListNode?) -> ListNode? {
        var current = head

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

        return head
    }
```

### Key Points:

**• Time: O(n)**
**• Space: O(1)**
**• Single pass**

### Notes:

*Skip duplicate nodes by updating next pointers.*

## A.3.8 Intersection of Two Linked Lists ■

Find the node where two linked lists intersect.

### Code Example:

```
func getIntersectionNode(_ headA: ListNode?, _ headB: ListNode?) -> ListNode? {
    var pointerA = headA
    var pointerB = headB

    while pointerA !== pointerB {
        pointerA = (pointerA == nil) ? headB : pointerA?.next
        pointerB = (pointerB == nil) ? headA : pointerB?.next
    }

    return pointerA
}
```

### Key Points:

**• Time: O(n + m)**
**• Space: O(1)**
**• Two pointers with switching**

### Notes:

*Switch to other list when reaching end; they meet at intersection.*

## A.3.9 Palindrome Linked List ■

Check if linked list forms a palindrome.

### Code Example:

```
func isPalindrome(_ head: ListNode?) -> Bool {
    guard let head = head else { return true }

    // Find middle using slow/fast pointers
    var slow = head
    var fast = head

    while fast.next != nil && fast.next?.next != nil {
        slow = slow.next!
        fast = fast.next!.next!
    }

    // Reverse second half
    var secondHalf = reverseList(slow.next)
    var firstHalf: ListNode? = head

    // Compare both halves
    while secondHalf != nil {
        if firstHalf!.val != secondHalf!.val {
            return false
        }
        firstHalf = firstHalf?.next
        secondHalf = secondHalf?.next
    }

    return true
}

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
}
```

### Key Points:

- **Time: O(n)**
- **Space: O(1)**
- **Find middle + reverse + compare**

### Notes:

*Find middle, reverse second half, compare with first half.*

## A.3.10 Add Two Numbers ■■

Add two numbers represented as linked lists.

### Code Example:

```
func addTwoNumbers(_ l1: ListNode?, _ l2: ListNode?) -> ListNode? {
    let dummy = ListNode(0)
    var current = dummy
    var carry = 0
    var p1 = l1, p2 = l2

    while p1 != nil || p2 != nil || carry > 0 {
        let sum = (p1?.val ?? 0) + (p2?.val ?? 0) + carry
        carry = sum / 10
        current.next = ListNode(sum % 10)
        current = current.next!

        p1 = p1?.next
        p2 = p2?.next
    }

    return dummy.next
}
```

### Key Points:

- **Time: O(max(n,m))**
- **Space: O(max(n,m))**
- **Digit by digit addition**

### Notes:

*Add digits with carry, handle different lengths and final carry.*

## A.3.11 Copy List with Random Pointer ■■

Deep copy a linked list with random pointers.

### Code Example:

```
class Node {
    var val: Int
    var next: Node?
    var random: Node?

    init(_ val: Int) {
        self.val = val
        self.next = nil
        self.random = nil
    }
}

func copyRandomList(_ head: Node?) -> Node? {
    guard let head = head else { return nil }

    var nodeMap: [Node: Node] = [:]
    var current: Node? = head

    // First pass: Create all nodes
    while let node = current {
        nodeMap[node] = Node(node.val)
        current = node.next
    }

    // Second pass: Set next and random pointers
    current = head
    while let node = current {
        nodeMap[node]?.next = nodeMap[node.next ?? Node(0)]
        nodeMap[node]?.random = nodeMap[node.random ?? Node(0)]
        current = node.next
    }

    return nodeMap[head]
}
```

### Key Points:

- **Time: O(n)**
- **Space: O(n)**
- **HashMap for node mapping**

### Notes:

*Use hashmap to map original nodes to copies, then set pointers.*

## A.3.12 LRU Cache ■■■

Implement Least Recently Used (LRU) cache.

### Code Example:

```
class LRUCache {
    class Node {
        var key: Int
        var value: Int
        var prev: Node?
        var next: Node?

        init(_ key: Int, _ value: Int) {
            self.key = key
            self.value = value
        }
    }

    private var capacity: Int
    private var cache: [Int: Node] = [:]
    private let head = Node(0, 0)
    private let tail = Node(0, 0)

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

    func get(_ key: Int) -> Int {
        if let node = cache[key] {
            moveToHead(node)
            return node.value
        }
        return -1
    }

    func put(_ key: Int, _ value: Int) {
        if let node = cache[key] {
            node.value = value
            moveToHead(node)
        } else {
            let newNode = Node(key, value)
            cache[key] = newNode
            addToHead(newNode)

            if cache.count > capacity {
                let removed = removeTail()
                cache.removeValue(forKey: removed.key)
            }
        }
    }

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

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

    private func moveToHead(_ node: Node) {
        removeNode(node)
        addToHead(node)
    }

    private func removeTail() -> Node {
        let last = tail.prev!
        removeNode(last)
        return last
    }
}
```

## Key Points:

- **Time: O(1) for get/put**
- **Space: O(capacity)**
- **HashMap + Doubly Linked List**

## Notes:

*Combine hashmap for O(1) access with doubly linked list for O(1) updates.*

# A.4 Binary Tree Problems (15 Problems)

```
// TreeNode Definition for Binary Tree Problems class TreeNode { var val: Int var
left: TreeNode? var right: TreeNode? init() { self.val = 0; self.left = nil;
self.right = nil } init(_ val: Int) { self.val = val; self.left = nil; self.right =
nil } init(_ val: Int, _ left: TreeNode?, _ right: TreeNode?) { self.val = val;
self.left = left; self.right = right } }
```

## A.4.1 Maximum Depth of Binary Tree ■

Find the maximum depth (height) of a binary tree.

### Code Example:

```
func maxDepth(_ root: TreeNode?) -> Int {
    guard let root = root else { return 0 }
    return 1 + max(maxDepth(root.left), maxDepth(root.right))
}

// Iterative BFS approach
func maxDepthIterative(_ root: TreeNode?) -> Int {
    guard let root = root else { return 0 }

    var queue: [TreeNode] = [root]
    var depth = 0

    while !queue.isEmpty {
        depth += 1
        let levelSize = queue.count

        for _ in 0..<levelSize {
            let node = queue.removeFirst()
            if let left = node.left { queue.append(left) }
            if let right = node.right { queue.append(right) }
        }
    }

    return depth
}
```

### Key Points:

**• Time: O(n)**
**• Space: O(h) recursive, O(w) iterative**
**• DFS/BFS**

### Notes:

*Use recursion or level-order traversal to count maximum depth.*

## A.4.2 Same Tree ■

Determine if two binary trees are identical.

### Code Example:

```
func isSameTree(_ p: TreeNode?, _ q: TreeNode?) -> Bool {
    if p == nil && q == nil { return true }
    if p == nil || q == nil { return false }

    return p!.val == q!.val &&
           isSameTree(p!.left, q!.left) &&
           isSameTree(p!.right, q!.right)
}

// Iterative approach using stacks
func isSameTreeIterative(_ p: TreeNode?, _ q: TreeNode?) -> Bool {
    var stack1: [TreeNode?] = [p]
    var stack2: [TreeNode?] = [q]

    while !stack1.isEmpty && !stack2.isEmpty {
        let node1 = stack1.removeLast()
        let node2 = stack2.removeLast()

        if node1 == nil && node2 == nil { continue }
        if node1 == nil || node2 == nil { return false }
        if node1!.val != node2!.val { return false }

        stack1.append(contentsOf: [node1!.left, node1!.right])
        stack2.append(contentsOf: [node2!.left, node2!.right])
    }

    return stack1.isEmpty && stack2.isEmpty
}
```

### Key Points:

- **Time: O(n)**
- **Space: O(h)**
- **DFS comparison**

### Notes:

*Recursively compare structure and values of both trees.*

## A.4.3 Invert Binary Tree ■

Invert (mirror) a binary tree.

### Code Example:

```
func invertTree(_ root: TreeNode?) -> TreeNode? {
        guard let root = root else { return nil }

        let temp = root.left
        root.left = invertTree(root.right)
        root.right = invertTree(temp)

        return root
    }

    // Iterative approach using queue
    func invertTreeIterative(_ root: TreeNode?) -> TreeNode? {
        guard let root = root else { return nil }

        var queue: [TreeNode] = [root]

        while !queue.isEmpty {
            let node = queue.removeFirst()

            // Swap children
            let temp = node.left
            node.left = node.right
            node.right = temp

            if let left = node.left { queue.append(left) }
            if let right = node.right { queue.append(right) }
        }

        return root
    }
```

### Key Points:

- **Time: O(n)**
- **Space: O(h)**
- **DFS with swapping**

### Notes:

*Swap left and right children recursively or iteratively.*

## A.4.4 Binary Tree Level Order Traversal ■■

Return level order traversal as array of arrays.

### Code Example:

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

        var result: [[Int]] = []
        var queue: [TreeNode] = [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
}
```

### Key Points:

- **Time: O(n)**
- **Space: O(w)**
- **BFS level by level**

### Notes:

*Use queue to process nodes level by level, collecting values.*

## A.4.5 Subtree of Another Tree ■■

Check if subRoot is a subtree of root.

### Code Example:

```
func isSubtree(_ root: TreeNode?, _ subRoot: TreeNode?) -> Bool {
    guard let root = root else { return subRoot == nil }

    return isSameTree(root, subRoot) ||
           isSubtree(root.left, subRoot) ||
           isSubtree(root.right, subRoot)
}

func isSameTree(_ p: TreeNode?, _ q: TreeNode?) -> Bool {
    if p == nil && q == nil { return true }
    if p == nil || q == nil { return false }

    return p!.val == q!.val &&
           isSameTree(p!.left, q!.left) &&
           isSameTree(p!.right, q!.right)
}
```

### Key Points:

- **Time: O(m*n)**

• **Space: O(h)**
• **DFS with tree comparison**

### Notes:

*Check if subtree matches at each node using isSameTree helper.*

## A.4.6 Lowest Common Ancestor ■■

Find LCA of two nodes in BST.

### Code Example:

```
func lowestCommonAncestor(_ root: TreeNode?, _ p: TreeNode?, _ q: TreeNode?) -> TreeNode? {
    guard let root = root, let p = p, let q = q else { return nil }

    // If both nodes are in left subtree
    if p.val < root.val && q.val < root.val {
        return lowestCommonAncestor(root.left, p, q)
    }

    // If both nodes are in right subtree
    if p.val > root.val && q.val > root.val {
        return lowestCommonAncestor(root.right, p, q)
    }

    // If nodes are on different sides or one equals root
    return root
}

// Iterative approach
func lowestCommonAncestorIterative(_ root: TreeNode?, _ p: TreeNode?, _ q: TreeNode?) -> Tre
    guard let p = p, let q = q else { return nil }
    var current = root

    while let node = current {
        if p.val < node.val && q.val < node.val {
            current = node.left
        } else if p.val > node.val && q.val > node.val {
            current = node.right
        } else {
            return node
        }
    }

    return nil
}
```

### Key Points:

• **Time: O(h)**
• **Space: O(h) recursive, O(1) iterative**
• **BST property**

### Notes:

*Use BST property to navigate left/right based on values.*

## A.4.7 Binary Tree Right Side View ■■

Return values of nodes you can see from the right side.

### Code Example:

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

    var result: [Int] = []
    var queue: [TreeNode] = [root]

    while !queue.isEmpty {
        let levelSize = queue.count

        for i in 0..<levelSize {
            let node = queue.removeFirst()

            // Add the rightmost node of each level
            if i == levelSize - 1 {
                result.append(node.val)
            }

            if let left = node.left { queue.append(left) }
            if let right = node.right { queue.append(right) }
        }
    }

    return result
}
```

### Key Points:

- **Time: O(n)**
- **Space: O(w)**
- **BFS level order**

### Notes:

*Use BFS and collect the rightmost node value from each level.*

## A.4.8 Count Good Nodes ■■

Count nodes where path from root contains no larger values.

### Code Example:

```
func goodNodes(_ root: TreeNode?) -> Int {
    return dfs(root, Int.min)
}

func dfs(_ node: TreeNode?, _ maxVal: Int) -> Int {
    guard let node = node else { return 0 }

    var count = 0
    if node.val >= maxVal {
        count = 1
    }

    let newMax = max(maxVal, node.val)
    count += dfs(node.left, newMax)
    count += dfs(node.right, newMax)

    return count
}
```

### Key Points:

**• Time: O(n)**
**• Space: O(h)**
**• DFS with max tracking**

### Notes:

*Use DFS to track maximum value in path and count good nodes.*

## A.4.9 Validate Binary Search Tree ■■

Determine if tree is a valid binary search tree.

### Code Example:

```
func isValidBST(_ root: TreeNode?) -> Bool {
    return validate(root, nil, nil)
}

func validate(_ node: TreeNode?, _ min: Int?, _ max: Int?) -> Bool {
    guard let node = node else { return true }

    if let min = min, node.val <= min { return false }
    if let max = max, node.val >= max { return false }

    return validate(node.left, min, node.val) &&
           validate(node.right, node.val, max)
}
```

```
// Alternative inorder approach
func isValidBSTInorder(_ root: TreeNode?) -> Bool {
    var prev: Int? = nil

    func inorder(_ node: TreeNode?) -> Bool {
        guard let node = node else { return true }

        if !inorder(node.left) { return false }

        if let prevVal = prev, node.val <= prevVal {
            return false
        }
        prev = node.val

        return inorder(node.right)
    }

    return inorder(root)
}
```

### Key Points:

- **Time: O(n)**
- **Space: O(h)**
- **Bounds checking or inorder**

### Notes:

*Use bounds checking or inorder traversal to validate BST property.*

## A.4.10 Kth Smallest in BST ■■

Find the kth smallest value in BST.

### Code Example:

```
func kthSmallest(_ root: TreeNode?, _ k: Int) -> Int {
    var count = 0
    var result = 0

    func inorder(_ node: TreeNode?) {
        guard let node = node, count < k else { return }

        inorder(node.left)

        count += 1
        if count == k {
            result = node.val
            return
        }

        inorder(node.right)
    }
    inorder(root)
    return result
}

// Iterative approach
func kthSmallestIterative(_ root: TreeNode?, _ k: Int) -> Int {
    var stack: [TreeNode] = []
    var current = root
    var count = 0

    while current != nil || !stack.isEmpty {
        while let node = current {
            stack.append(node)
            current = node.left
        }

        current = stack.removeLast()
        count += 1

        if count == k {
            return current!.val
        }

        current = current?.right
    }

    return -1 // Should never reach here if k is valid
}
```

### Key Points:

- **Time: O(h + k)**
- **Space: O(h)**
- **Inorder traversal**

### Notes:

*Use inorder traversal to visit nodes in sorted order, return kth.*

## A.4.11 Construct Tree from Traversals ■■

Build binary tree from preorder and inorder traversal.

### Code Example:

```
func buildTree(_ preorder: [Int], _ inorder: [Int]) -> TreeNode? {
    guard !preorder.isEmpty && !inorder.isEmpty else { return nil }

    let root = TreeNode(preorder[0])
    guard let mid = inorder.firstIndex(of: preorder[0]) else { return root }

    root.left = buildTree(Array(preorder[1..<mid+1]), Array(inorder[0..<mid]))
    root.right = buildTree(Array(preorder[mid+1..<preorder.count]), Array(inorder[mid+1..<in
    return root
}
```

### Key Points:

**• Time: O(n)**
**• Space: O(n)**
**• Recursive construction**

### Notes:

*Use preorder for root, inorder to split left/right subtrees.*

## A.4.12 Binary Tree Max Path Sum ■■■

Find the maximum path sum in binary tree.

### Code Example:

```
func maxPathSum(_ root: TreeNode?) -> Int {
    var maxSum = Int.min

    func maxGain(_ node: TreeNode?) -> Int {
        guard let node = node else { return 0 }

        let leftGain = max(maxGain(node.left), 0)
        let rightGain = max(maxGain(node.right), 0)

        let currentPathSum = node.val + leftGain + rightGain
        maxSum = max(maxSum, currentPathSum)

        return node.val + max(leftGain, rightGain)
    }

    _ = maxGain(root)
    return maxSum
}
```

### Key Points:

**• Time: O(n)**
**• Space: O(h)**
**• DFS with global maximum**

### Notes:

*Calculate max gain from each subtree, track global maximum path sum.*

## A.4.13 Serialize and Deserialize Tree ■■■

Encode tree to string and decode string to tree.

## Code Example:

```
class Codec {
    func serialize(_ root: TreeNode?) -> String {
        var result: [String] = []

        func preorder(_ node: TreeNode?) {
            if let node = node {
                result.append(String(node.val))
                preorder(node.left)
                preorder(node.right)
            } else {
                result.append("null")
            }
        }

        preorder(root)
        return result.joined(separator: ",")
    }

    func deserialize(_ data: String) -> TreeNode? {
        let values = data.split(separator: ",").map { String($0) }
        var index = 0

        func buildTree() -> TreeNode? {
            guard index < values.count else { return nil }

            let val = values[index]
            index += 1

            if val == "null" { return nil }

            let node = TreeNode(Int(val)!)
            node.left = buildTree()
            node.right = buildTree()

            return node
        }

        return buildTree()
    }
}
```

## Key Points:

- **Time: O(n)**
- **Space: O(n)**
- **Preorder with null markers**

## Notes:

*Use preorder traversal with null markers for serialization.*

## A.4.14 Word Search II ■■■

Find all words from word list that can be constructed on board.

### Code Example:

```
class TrieNode {
    var children: [Character: TrieNode] = [:]
    var word: String? = nil
}

func findWords(_ board: [[Character]], _ words: [String]) -> [String] {
    // Build Trie
    let root = TrieNode()
    for word in words {
        var node = root
        for char in word {
            if node.children[char] == nil {
                node.children[char] = TrieNode()
            }
            node = node.children[char]!
        }
        node.word = word
    }

    var result: Set<String> = []
    var board = board

    func dfs(_ row: Int, _ col: Int, _ node: TrieNode) {
        guard row >= 0 && row < board.count &&
              col >= 0 && col < board[0].count else { return }

        let char = board[row][col]
        guard char != "#", let nextNode = node.children[char] else { return }

        if let word = nextNode.word {
            result.insert(word)
        }

        board[row][col] = "#" // mark visited

        dfs(row + 1, col, nextNode)
        dfs(row - 1, col, nextNode)
        dfs(row, col + 1, nextNode)
        dfs(row, col - 1, nextNode)

        board[row][col] = char // backtrack
    }

    for i in 0..<board.count {
        for j in 0..<board[0].count {
            dfs(i, j, root)
        }
    }

    return Array(result)
}
```

### Key Points:

- **Time: O(m*n*4^L)**
- **Space: O(N)**
- **Trie + DFS backtracking**

### Notes:

*Build Trie from words, DFS on board with backtracking.*

## A.4.15 Balanced Binary Tree ■

Check if binary tree is height-balanced.

### Code Example:

```
func isBalanced(_ root: TreeNode?) -> Bool {
    return checkHeight(root) != -1
}

func checkHeight(_ node: TreeNode?) -> Int {
    guard let node = node else { return 0 }

    let leftHeight = checkHeight(node.left)
    if leftHeight == -1 { return -1 }

    let rightHeight = checkHeight(node.right)
    if rightHeight == -1 { return -1 }

    if abs(leftHeight - rightHeight) > 1 {
        return -1
    }

    return 1 + max(leftHeight, rightHeight)
}
```

### Key Points:

- **Time: O(n)**
- **Space: O(h)**
- **Bottom-up height checking**

### Notes:

*Calculate heights bottom-up, return -1 if unbalanced found.*

# A.5 Dynamic Programming Problems (15 Problems)

## A.5.1 Climbing Stairs ■

Count ways to reach nth stair (1 or 2 steps).

### Code Example:

```
func climbStairs(_ n: Int) -> Int {
    guard n > 2 else { return n }

    var dp = [0, 1, 2]

    for i in 3...n {
        let ways = dp[1] + dp[2]
        dp[1] = dp[2]
        dp[2] = ways
    }

    return dp[2]
}

// Recursive with memoization
func climbStairsRecursive(_ n: Int) -> Int {
    var memo: [Int: Int] = [:]

    func climb(_ step: Int) -> Int {
        if step <= 2 { return step }

        if let cached = memo[step] {
            return cached
        }

        memo[step] = climb(step - 1) + climb(step - 2)
        return memo[step]!
    }

    return climb(n)
}
```

### Key Points:

**• Time: O(n)**
**• Space: O(1) iterative, O(n) recursive**
**• Fibonacci pattern DP**

### Notes:

*Each step can be reached from previous step or two steps before.*

## A.5.2 House Robber ■■

Rob houses without robbing adjacent ones.

### Code Example:

```
func rob(_ nums: [Int]) -> Int {
    guard !nums.isEmpty else { return 0 }
    guard nums.count > 1 else { return nums[0] }

    var prev1 = 0  // max money up to i-1
    var prev2 = 0  // max money up to i-2

    for num in nums {
        let current = max(prev1, prev2 + num)
        prev2 = prev1
        prev1 = current
    }

    return prev1
}

// Alternative with explicit DP array
func robDP(_ nums: [Int]) -> Int {
    guard !nums.isEmpty else { return 0 }
    guard nums.count > 1 else { return nums[0] }

    var dp = Array(repeating: 0, count: nums.count)
    dp[0] = nums[0]
    dp[1] = max(nums[0], nums[1])

    for i in 2..<nums.count {
        dp[i] = max(dp[i-1], dp[i-2] + nums[i])
    }

    return dp[nums.count - 1]
}
```

### Key Points:

**• Time: O(n)**
**• Space: O(1)**
**• Linear DP optimization**

### Notes:

*Track maximum money without robbing adjacent houses.*

## A.5.3 House Robber II ■■

Rob houses in circular array (first and last adjacent).

## Code Example:

```
func robCircular(_ nums: [Int]) -> Int {
    guard !nums.isEmpty else { return 0 }
    guard nums.count > 1 else { return nums[0] }
    guard nums.count > 2 else { return max(nums[0], nums[1]) }

    func robLinear(_ houses: [Int]) -> Int {
        var prev1 = 0, prev2 = 0

        for house in houses {
            let current = max(prev1, prev2 + house)
            prev2 = prev1
            prev1 = current
        }

        return prev1
    }

    // Case 1: Rob first house, skip last (houses[0...n-2])
    let robWithFirst = robLinear(Array(nums[0..<nums.count-1]))

    // Case 2: Skip first house, can rob last (houses[1...n-1])
    let robWithoutFirst = robLinear(Array(nums[1..<nums.count]))

    return max(robWithFirst, robWithoutFirst)
}
```

## Key Points:

**• Time: O(n)**
**• Space: O(1)**
**• Circular array DP**

## Notes:

*Handle circular constraint by solving two linear cases.*

## A.5.4 Longest Palindromic Subsequence ■■

Find length of longest palindromic subsequence.

## Code Example:

```
func longestPalindromeSubseq(_ s: String) -> Int {
    let chars = Array(s)
    let n = chars.count
    var dp = Array(repeating: Array(repeating: 0, count: n), count: n)

    // Every single character is palindrome of length 1
    for i in 0..<n {
        dp[i][i] = 1
    }

    // Check for subsequences of length 2 to n
    for length in 2...n {
        for i in 0...(n - length) {
            let j = i + length - 1

            if chars[i] == chars[j] {
                dp[i][j] = dp[i+1][j-1] + 2
            } else {
                dp[i][j] = max(dp[i+1][j], dp[i][j-1])
            }
        }
    }

    return dp[0][n-1]
}
```

### Key Points:

- **Time: O(n²)**
- **Space: O(n²)**
- **2D DP approach**

### Notes:

*Build table bottom-up, expand from center matches.*

## A.5.5 Palindromic Substrings ■■

Count all palindromic substrings in string.

### Code Example:

```
func countSubstrings(_ s: String) -> Int {
    let chars = Array(s)
    var count = 0

    func expandAroundCenter(_ left: Int, _ right: Int) -> Int {
        var l = left, r = right, palindromes = 0

        while l >= 0 && r < chars.count && chars[l] == chars[r] {
            palindromes += 1
            l -= 1
            r += 1
        }

        return palindromes
    }

    for i in 0..<chars.count {
        // Odd length palindromes (center at i)
        count += expandAroundCenter(i, i)

        // Even length palindromes (center between i and i+1)
        count += expandAroundCenter(i, i + 1)
    }

    return count
}

// Alternative: DP approach
func countSubstringsDP(_ s: String) -> Int {
    let chars = Array(s)
    let n = chars.count
    var dp = Array(repeating: Array(repeating: false, count: n), count: n)
    var count = 0

    // Every single character is palindrome
    for i in 0..<n {
        dp[i][i] = true
        count += 1
    }

    // Check for length 2
    for i in 0..<n-1 {
        if chars[i] == chars[i+1] {
            dp[i][i+1] = true
            count += 1
        }
    }

    // Check for lengths 3 to n
    for length in 3...n {
        for i in 0...(n-length) {
            let j = i + length - 1

            if chars[i] == chars[j] && dp[i+1][j-1] {
                dp[i][j] = true
                count += 1
            }
        }
    }

    return count
}
```

## Key Points:

- **Time: O($n^2$)**
- **Space: O(1) expand, O($n^2$) DP**
- **Expand around centers**

## Notes:

*Check all possible centers for palindromes.*

### A.5.6 Decode Ways ■■

Count ways to decode string

#### Code Example:

```
func numDecodings(_ s: String) -> Int {
    let chars = Array(s)
    guard !chars.isEmpty && chars[0] != "0" else { return 0 }

    var dp1 = 1, dp2 = 1  // dp[i-1], dp[i-2]

    for i in 1..<chars.count {
        var current = 0

        if chars[i] != "0" {
            current += dp1
        }

        let twoDigit = Int(String(chars[i-1...i]))!
        if twoDigit >= 10 && twoDigit <= 26 {
            current += dp2
        }

        dp2 = dp1
        dp1 = current
    }

    return dp1
}
```

#### Key Points:

- **Time: O(n)**
- **Space: O(1)**

#### Notes:

*String DP with 1/2 digit decoding*

### A.5.7 Coin Change ■■

Find minimum coins for amount

#### Code Example:

```
func coinChange(_ coins: [Int], _ amount: Int) -> Int {
    var dp = Array(repeating: amount + 1, count: amount + 1)
    dp[0] = 0
    for i in 1...amount {
        for coin in coins {
            if coin <= i {
                dp[i] = min(dp[i], dp[i - coin] + 1)
            }
        }
    }

    return dp[amount] > amount ? -1 : dp[amount]
}
```

### Key Points:

- **Time: O(amount × coins)**
- **Space: O(amount)**

### Notes:

*Unbounded knapsack DP*

## A.5.8 Maximum Product Subarray ■■

Find max product subarray

### Code Example:

```
func maxProduct(_ nums: [Int]) -> Int {
    var maxProd = nums[0], minProd = nums[0], result = nums[0]

    for i in 1..<nums.count {
        let num = nums[i]
        let tempMax = max(num, num * maxProd, num * minProd)
        minProd = min(num, num * maxProd, num * minProd)
        maxProd = tempMax
        result = max(result, maxProd)
    }

    return result
}
```

### Key Points:

- **Time: O(n)**
- **Space: O(1)**

### Notes:

*Track min/max products for negatives*

### A.5.9 Word Break ■■

Check if string can be segmented

#### Code Example:

```
func wordBreak(_ s: String, _ wordDict: [String]) -> Bool {
    let wordSet = Set(wordDict)
    var dp = Array(repeating: false, count: s.count + 1)
    dp[0] = true

    let chars = Array(s)

    for i in 1...s.count {
        for j in 0..<i {
            if dp[j] && wordSet.contains(String(chars[j..<i])) {
                dp[i] = true
                break
            }
        }
    }

    return dp[s.count]
}
```

#### Key Points:

- **Time: O($n^3$)**
- **Space: O(n)**

#### Notes:

*String segmentation DP*

### A.5.10 Combination Sum IV ■■

Count combinations for target

#### Code Example:

```
func combinationSum4(_ nums: [Int], _ target: Int) -> Int {
    var dp = Array(repeating: 0, count: target + 1)
    dp[0] = 1

    for i in 1...target {
        for num in nums {
            if num <= i {
                dp[i] += dp[i - num]
            }
        }
    }
```

```
        return dp[target]
    }
```

### Key Points:

**• Time: O(target × n)**
**• Space: O(target)**

### Notes:

*Count combinations DP*

## A.5.11 Longest Increasing Subsequence ■■

Find LIS length

### Code Example:

```
    func lengthOfLIS(_ nums: [Int]) -> Int {
        var tails: [Int] = []

        for num in nums {
            var left = 0, right = tails.count

            while left < right {
                let mid = (left + right) / 2
                if tails[mid] < num {
                    left = mid + 1
                } else {
                    right = mid
                }
            }

            if left == tails.count {
                tails.append(num)
            } else {
                tails[left] = num
            }
        }

        return tails.count
    }
```

### Key Points:

**• Time: O(n log n)**
**• Space: O(n)**

### Notes:

*LIS with binary search*

### A.5.12 Unique Paths ■■

Count paths in grid

#### Code Example:

```
func uniquePaths(_ m: Int, _ n: Int) -> Int {
    var dp = Array(repeating: 1, count: n)

    for i in 1..<m {
        for j in 1..<n {
            dp[j] += dp[j-1]
        }
    }

    return dp[n-1]
}
```

#### Key Points:

- **Time: O(m×n)**
- **Space: O(n)**

#### Notes:

*Grid path counting DP*

### A.5.13 Jump Game ■■

Check if can reach end

#### Code Example:

```
func canJump(_ nums: [Int]) -> Bool {
    var maxReach = 0

    for i in 0..<nums.count {
        if i > maxReach { return false }
        maxReach = max(maxReach, i + nums[i])
    }

    return true
}
```

#### Key Points:

- **Time: O(n)**
- **Space: O(1)**

### Notes:

*Greedy approach better than DP*

## A.5.14 Edit Distance ■■■

Minimum edit operations

### Code Example:

```
func minDistance(_ word1: String, _ word2: String) -> Int {
    let chars1 = Array(word1), chars2 = Array(word2)
    let m = chars1.count, n = chars2.count

    var dp = Array(repeating: Array(repeating: 0, count: n + 1), count: m + 1)

    for i in 0...m { dp[i][0] = i }
    for j in 0...n { dp[0][j] = j }

    for i in 1...m {
        for j in 1...n {
            if chars1[i-1] == chars2[j-1] {
                dp[i][j] = dp[i-1][j-1]
            } else {
                dp[i][j] = 1 + min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1])
            }
        }
    }

    return dp[m][n]
}
```

### Key Points:

**• Time: O(m×n)**
**• Space: O(m×n)**

### Notes:

*Levenshtein distance DP*

## A.5.15 Regular Expression Matching ■■■

Pattern matching with . and *

### Code Example:

```
func isMatch(_ s: String, _ p: String) -> Bool {
    let sChars = Array(s), pChars = Array(p)
```

```
        let m = sChars.count, n = pChars.count

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

        for j in 2...n {
            if pChars[j-1] == "*" {
                dp[0][j] = dp[0][j-2]
            }
        }

        for i in 1...m {
            for j in 1...n {
                let sc = sChars[i-1], pc = pChars[j-1]

                if pc == "*" {
                    dp[i][j] = dp[i][j-2]  // Match 0 times
                    if sChars[i-1] == pChars[j-2] || pChars[j-2] == "." {
                        dp[i][j] = dp[i][j] || dp[i-1][j]  // Match 1+ times
                    }
                } else if sc == pc || pc == "." {
                    dp[i][j] = dp[i-1][j-1]
                }
            }
        }

        return dp[m][n]
    }
```

## Key Points:

- **Time: O(m×n)**
- **Space: O(m×n)**

## Notes:

*2D DP with pattern matching*

# A.6 Graph Problems (12 Problems)

## A.6.1 Number of Islands ■■

Count distinct islands in 2D grid using DFS.

### Code Example:

```
func numIslands(_ grid: [[Character]]) -> Int {
    guard !grid.isEmpty else { return 0 }

    var grid = grid
    var count = 0

    func dfs(_ row: Int, _ col: Int) {
        guard row >= 0 && row < grid.count &&
              col >= 0 && col < grid[0].count &&
              grid[row][col] == "1" else { return }

        grid[row][col] = "0"  // Mark as visited

        dfs(row + 1, col)  // Down
        dfs(row - 1, col)  // Up
        dfs(row, col + 1)  // Right
        dfs(row, col - 1)  // Left
    }

    for i in 0..<grid.count {
        for j in 0..<grid[0].count {
            if grid[i][j] == "1" {
                count += 1
                dfs(i, j)
            }
        }
    }

    return count
}
```

### Key Points:

- **Time: O(m×n)**
- **Space: O(m×n)**
- **DFS traversal**

### Notes:

*Use DFS to explore and mark connected land cells for each island.*

## A.6.2 Clone Graph ■■

Deep clone an undirected graph.

### Code Example:

```
class Node {
    public var val: Int
    public var neighbors: [Node?]
    public init(_ val: Int) {
        self.val = val
        self.neighbors = []
    }
}

func cloneGraph(_ node: Node?) -> Node? {
    guard let node = node else { return nil }

    var visited: [Int: Node] = [:]

    func dfs(_ node: Node) -> Node {
        if let clone = visited[node.val] {
            return clone
        }

        let clone = Node(node.val)
        visited[node.val] = clone

        for neighbor in node.neighbors {
            if let neighbor = neighbor {
                clone.neighbors.append(dfs(neighbor))
            }
        }

        return clone
    }

    return dfs(node)
}
```

### Key Points:

- **Time: O(V + E)**
- **Space: O(V)**
- **DFS with hashmap**

### Notes:

*Use DFS with hashmap to track cloned nodes and avoid cycles.*

## A.6.3 Max Area of Island ■■

Find the area of the largest island.

### Code Example:

```
func maxAreaOfIsland(_ grid: [[Int]]) -> Int {
    guard !grid.isEmpty else { return 0 }

    var grid = grid
    var maxArea = 0

    func dfs(_ row: Int, _ col: Int) -> Int {
        guard row >= 0 && row < grid.count &&
              col >= 0 && col < grid[0].count &&
              grid[row][col] == 1 else { return 0 }

        grid[row][col] = 0  // Mark as visited

        return 1 + dfs(row + 1, col) + dfs(row - 1, col) +
                   dfs(row, col + 1) + dfs(row, col - 1)
    }

    for i in 0..<grid.count {
        for j in 0..<grid[0].count {
            if grid[i][j] == 1 {
                maxArea = max(maxArea, dfs(i, j))
            }
        }
    }

    return maxArea
}
```

### Key Points:

**• Time: O(m×n)**
**• Space: O(m×n)**
**• DFS with area calculation**

### Notes:

*Use DFS to calculate area of each island, track maximum.*

## A.6.4 Pacific Atlantic Water Flow ■■

Find cells from which water can flow to both oceans.

### Code Example:

```
func pacificAtlantic(_ heights: [[Int]]) -> [[Int]] {
    let m = heights.count, n = heights[0].count
    var pacific = Array(repeating: Array(repeating: false, count: n), count: m)
    var atlantic = Array(repeating: Array(repeating: false, count: n), count: m)
```

```
        func dfs(_ row: Int, _ col: Int, _ visited: inout [[Bool]], _ prevHeight: Int) {
            guard row >= 0 && row < m && col >= 0 && col < n &&
                  !visited[row][col] && heights[row][col] >= prevHeight else { return }

            visited[row][col] = true

            dfs(row + 1, col, &visited, heights[row][col])
            dfs(row - 1, col, &visited, heights[row][col])
            dfs(row, col + 1, &visited, heights[row][col])
            dfs(row, col - 1, &visited, heights[row][col])
        }

        // DFS from Pacific border
        for i in 0..<m {
            dfs(i, 0, &pacific, heights[i][0])
        }
        for j in 0..<n {
            dfs(0, j, &pacific, heights[0][j])
        }

        // DFS from Atlantic border
        for i in 0..<m {
            dfs(i, n - 1, &atlantic, heights[i][n - 1])
        }
        for j in 0..<n {
            dfs(m - 1, j, &atlantic, heights[m - 1][j])
        }

        var result: [[Int]] = []
        for i in 0..<m {
            for j in 0..<n {
                if pacific[i][j] && atlantic[i][j] {
                    result.append([i, j])
                }
            }
        }

        return result
    }
```

### Key Points:

- **Time: O(m×n)**
- **Space: O(m×n)**
- **Reverse DFS from borders**

### Notes:

*Start DFS from ocean borders, find cells reachable by both.*

## A.6.5 Surrounded Regions ■■

Capture surrounded regions using boundary DFS.

### Code Example:

```swift
func solve(_ board: inout [[Character]]) {
    guard !board.isEmpty else { return }

    let m = board.count, n = board[0].count

    func dfs(_ row: Int, _ col: Int) {
        guard row >= 0 && row < m && col >= 0 && col < n &&
              board[row][col] == "O" else { return }

        board[row][col] = "#"  // Mark as boundary-connected

        dfs(row + 1, col)
        dfs(row - 1, col)
        dfs(row, col + 1)
        dfs(row, col - 1)
    }

    // DFS from all boundary O's
    for i in 0..<m {
        dfs(i, 0)      // Left border
        dfs(i, n - 1)  // Right border
    }
    for j in 0..<n {
        dfs(0, j)      // Top border
        dfs(m - 1, j)  // Bottom border
    }

    // Convert remaining O's to X's, and #'s back to O's
    for i in 0..<m {
        for j in 0..<n {
            if board[i][j] == "O" {
                board[i][j] = "X"
            } else if board[i][j] == "#" {
                board[i][j] = "O"
            }
        }
    }
}
```

### Key Points:

- **Time: O(m×n)**
- **Space: O(m×n)**
- **Boundary DFS technique**

### Notes:

*Start DFS from borders to identify boundary-connected regions.*

## A.6.6 Rotting Oranges ■■

Find time for all oranges to rot using multi-source BFS.

## Code Example:

```
func orangesRotting(_ grid: [[Int]]) -> Int {
    let m = grid.count, n = grid[0].count
    var grid = grid
    var queue: [(Int, Int)] = []
    var freshCount = 0

    // Find initial rotten oranges and count fresh ones
    for i in 0..<m {
        for j in 0..<n {
            if grid[i][j] == 2 {
                queue.append((i, j))
            } else if grid[i][j] == 1 {
                freshCount += 1
            }
        }
    }

    guard freshCount > 0 else { return 0 }

    let directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    var minutes = 0

    while !queue.isEmpty && freshCount > 0 {
        let levelSize = queue.count

        for _ in 0..<levelSize {
            let (row, col) = queue.removeFirst()

            for (dr, dc) in directions {
                let newRow = row + dr
                let newCol = col + dc

                if newRow >= 0 && newRow < m && newCol >= 0 && newCol < n &&
                   grid[newRow][newCol] == 1 {
                    grid[newRow][newCol] = 2
                    queue.append((newRow, newCol))
                    freshCount -= 1
                }
            }
        }

        minutes += 1
    }

    return freshCount == 0 ? minutes : -1
}
```

## Key Points:

**• Time: O(m×n)**
**• Space: O(m×n)**
**• Multi-source BFS**

### Notes:

*Start BFS from all initial rotten oranges simultaneously.*

## A.6.7 Course Schedule ■■

Detect if course schedule is possible (no cycles).

### Code Example:

```
func canFinish(_ numCourses: Int, _ prerequisites: [[Int]]) -> Bool {
    var graph: [Int: [Int]] = [:]
    var inDegree = Array(repeating: 0, count: numCourses)

    // Build graph and calculate in-degrees
    for prereq in prerequisites {
        let course = prereq[0]
        let prerequisite = prereq[1]

        graph[prerequisite, default: []].append(course)
        inDegree[course] += 1
    }

    // Kahn's algorithm for topological sort
    var queue: [Int] = []
    for i in 0..<numCourses {
        if inDegree[i] == 0 {
            queue.append(i)
        }
    }

    var processedCourses = 0

    while !queue.isEmpty {
        let course = queue.removeFirst()
        processedCourses += 1

        for neighbor in graph[course, default: []] {
            inDegree[neighbor] -= 1
            if inDegree[neighbor] == 0 {
                queue.append(neighbor)
            }
        }
    }

    return processedCourses == numCourses
}
```

### Key Points:

**• Time: O(V + E)**
**• Space: O(V + E)**
**• Topological sort - cycle detection**

### Notes:

*Use Kahn's algorithm to detect cycles in course dependency graph.*

## A.6.8 Course Schedule II ■■

Return course order if possible (topological ordering).

### Code Example:

```
func findOrder(_ numCourses: Int, _ prerequisites: [[Int]]) -> [Int] {
    var graph: [Int: [Int]] = [:]
    var inDegree = Array(repeating: 0, count: numCourses)

    // Build graph and calculate in-degrees
    for prereq in prerequisites {
        let course = prereq[0]
        let prerequisite = prereq[1]

        graph[prerequisite, default: []].append(course)
        inDegree[course] += 1
    }

    // Kahn's algorithm for topological sort
    var queue: [Int] = []
    for i in 0..<numCourses {
        if inDegree[i] == 0 {
            queue.append(i)
        }
    }

    var result: [Int] = []

    while !queue.isEmpty {
        let course = queue.removeFirst()
        result.append(course)

        for neighbor in graph[course, default: []] {
            inDegree[neighbor] -= 1
            if inDegree[neighbor] == 0 {
                queue.append(neighbor)
            }
        }
    }

    return result.count == numCourses ? result : []
}
```

### Key Points:

- **Time: O(V + E)**
- **Space: O(V + E)**
- **Topological ordering**

### Notes:

*Use Kahn's algorithm to generate valid course sequence.*

## A.6.9 Redundant Connection ■■

Find redundant edge that creates cycle using Union-Find.

### Code Example:

```
func findRedundantConnection(_ edges: [[Int]]) -> [Int] {
    let n = edges.count
    var parent = Array(0...n)

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

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

        if rootX == rootY {
            return false  // Already connected (cycle found)
        }

        parent[rootX] = rootY
        return true
    }

    for edge in edges {
        if !union(edge[0], edge[1]) {
            return edge  // This edge creates cycle
        }
    }

    return []
}
```

### Key Points:

- **Time: O(n α(n))**
- **Space: O(n)**
- **Union-Find with path compression**

### Notes:

*Use Union-Find to detect when adding edge creates cycle.*

## A.6.10 Word Ladder ■■■

Find shortest transformation path using BFS.

### Code Example:

```
func ladderLength(_ beginWord: String, _ endWord: String, _ wordList: [String]) -> Int {
    var wordSet = Set(wordList)
    guard wordSet.contains(endWord) else { return 0 }

    var queue = [(beginWord, 1)]

    while !queue.isEmpty {
        let (currentWord, steps) = queue.removeFirst()

        if currentWord == endWord {
            return steps
        }

        let wordArray = Array(currentWord)

        for i in 0..<wordArray.count {
            for c in "abcdefghijklmnopqrstuvwxyz" {
                if c == wordArray[i] { continue }

                var newWordArray = wordArray
                newWordArray[i] = c
                let newWord = String(newWordArray)

                if wordSet.contains(newWord) {
                    if newWord == endWord {
                        return steps + 1
                    }
                    wordSet.remove(newWord)
                    queue.append((newWord, steps + 1))
                }
            }
        }
    }

    return 0
}
```

### Key Points:

**• Time: $O(m^2 \times n)$**
**• Space: $O(m \times n)$**
**• BFS with string transformation**

### Notes:

*BFS to find shortest path by trying all single-character changes.*

## A.6.11 Alien Dictionary ■■■

Determine character order using topological sort.

### Code Example:

```
func alienOrder(_ words: [String]) -> String {
    var graph: [Character: Set<Character>] = [:]
    var inDegree: [Character: Int] = [:]

    // Initialize all characters
    for word in words {
        for char in word {
            graph[char] = []
            inDegree[char] = 0
        }
    }

    // Build graph from adjacent words
    for i in 0..<words.count - 1 {
        let word1 = Array(words[i])
        let word2 = Array(words[i + 1])

        let minLength = min(word1.count, word2.count)

        for j in 0..<minLength {
            if word1[j] != word2[j] {
                if !graph[word1[j]]!.contains(word2[j]) {
                    graph[word1[j]]!.insert(word2[j])
                    inDegree[word2[j]]! += 1
                }
                break
            }
        }

        // Check invalid case: word2 is prefix of word1
        if word1.count > word2.count &&
           Array(word1.prefix(word2.count)) == word2 {
            return ""
        }
    }

    // Topological sort using Kahn's algorithm
    var queue: [Character] = []
    for (char, degree) in inDegree {
        if degree == 0 {
            queue.append(char)
        }
    }

    var result = ""

    while !queue.isEmpty {
        let char = queue.removeFirst()
        result.append(char)
        for neighbor in graph[char]! {
            inDegree[neighbor]! -= 1
            if inDegree[neighbor]! == 0 {
                queue.append(neighbor)
            }
        }
    }

    return result.count == inDegree.count ? result : ""
}
```

### Key Points:

- **Time: O(c)**
- **Space: O(1)**
- **Topological sort with character ordering**

### Notes:

*Build dependency graph from word comparisons, use topological sort.*

## A.6.12 Network Delay Time ■■

Find minimum time to reach all nodes using Dijkstra's algorithm.

### Code Example:

```
func networkDelayTime(_ times: [[Int]], _ n: Int, _ k: Int) -> Int {
    var graph: [Int: [(Int, Int)]] = [:]

    // Build adjacency list
    for time in times {
        let u = time[0], v = time[1], w = time[2]
        graph[u, default: []].append((v, w))
    }

    // Dijkstra's algorithm
    var distances = Array(repeating: Int.max, count: n + 1)
    distances[k] = 0

    var heap: [(Int, Int)] = [(0, k)]  // (distance, node)

    while !heap.isEmpty {
        heap.sort { $0.0 < $1.0 }
        let (currentDistance, node) = heap.removeFirst()

        if currentDistance > distances[node] {
            continue
        }

        for (neighbor, weight) in graph[node, default: []] {
            let newDistance = currentDistance + weight
```

```
                if newDistance < distances[neighbor] {
                    distances[neighbor] = newDistance
                    heap.append((newDistance, neighbor))
                }
            }
        }

        let maxDistance = distances[1...n].max()!
        return maxDistance == Int.max ? -1 : maxDistance
    }
```

## Key Points:

**• Time: O(E log V)**
**• Space: O(V + E)**
**• Dijkstra's shortest path**

## Notes:

*Use Dijkstra's algorithm to find shortest paths from source to all nodes.*

## A.7 Heap & Priority Queue Problems (8 Problems)

### A.7.1 Kth Largest Element ■■

Find the kth largest element in array.

#### Code Example:

```
func findKthLargest(_ nums: [Int], _ k: Int) -> Int {
    var heap = nums.prefix(k).sorted()  // Min heap simulation

    for i in k..<nums.count {
        if nums[i] > heap[0] {
            heap[0] = nums[i]
            heap.sort()  // Re-heapify
        }
    }

    return heap[0]
}

// Quick Select approach (more efficient)
func findKthLargestQuickSelect(_ nums: [Int], _ k: Int) -> Int {
    var nums = nums
    let target = nums.count - k

    func partition(_ left: Int, _ right: Int) -> Int {
        let pivot = nums[right]
        var i = left

        for j in left..<right {
            if nums[j] <= pivot {
                nums.swapAt(i, j)
                i += 1
            }
        }
        nums.swapAt(i, right)
        return i
    }

    func quickSelect(_ left: Int, _ right: Int) -> Int {
        let pivotIndex = partition(left, right)

        if pivotIndex == target {
            return nums[pivotIndex]
        } else if pivotIndex < target {
            return quickSelect(pivotIndex + 1, right)
        } else {
            return quickSelect(left, pivotIndex - 1)
        }
    }

    return quickSelect(0, nums.count - 1)
}
```

### Key Points:

- **Time: O(n) average QuickSelect, O(n log k) heap**
- **Space: O(k)**
- **Quick select or min heap**

### Notes:

*Use quickselect for O(n) average, or maintain min heap of size k.*

## A.7.2 Last Stone Weight ■

Simulate stone smashing with max heap.

### Code Example:

```
    func lastStoneWeight(_ stones: [Int]) -> Int {
        var maxHeap = stones.sorted(by: >)

        while maxHeap.count > 1 {
            let first = maxHeap.removeFirst()
            let second = maxHeap.removeFirst()

            if first != second {
                let newStone = first - second
                // Insert in sorted order (maintaining max heap property)
                var inserted = false
                for i in 0..<maxHeap.count {
                    if newStone > maxHeap[i] {
                        maxHeap.insert(newStone, at: i)
                        inserted = true
                        break
                    }
                }
                if !inserted {
                    maxHeap.append(newStone)
                }
            }
        }

        return maxHeap.first ?? 0
    }
```

### Key Points:

- **Time: O($n^2$ log n)**
- **Space: O(1)**

**• Max heap simulation**

## Notes:

*Use max heap to always get the two heaviest stones.*

## A.7.3 K Closest Points to Origin ■■

Find k closest points to origin using heap.

### Code Example:

```
func kClosest(_ points: [[Int]], _ k: Int) -> [[Int]] {
    func distance(_ point: [Int]) -> Int {
        return point[0] * point[0] + point[1] * point[1]
    }

    // Use max heap to maintain k closest points
    var maxHeap: [(dist: Int, point: [Int])] = []

    for point in points {
        let dist = distance(point)

        if maxHeap.count < k {
            maxHeap.append((dist, point))
            maxHeap.sort { $0.dist > $1.dist }
        } else if dist < maxHeap[0].dist {
            maxHeap[0] = (dist, point)
            maxHeap.sort { $0.dist > $1.dist }
        }
    }

    return maxHeap.map { $0.point }
}

// Alternative: Sort approach
func kClosestSort(_ points: [[Int]], _ k: Int) -> [[Int]] {
    let sortedPoints = points.sorted {
        $0[0]*$0[0] + $0[1]*$0[1] < $1[0]*$1[0] + $1[1]*$1[1]
    }
    return Array(sortedPoints.prefix(k))
}
```

### Key Points:

**• Time: O(n log k) heap, O(n log n) sort**
**• Space: O(k)**
**• Max heap or sorting**

### Notes:

*Use max heap to keep k smallest distances, or sort all points.*

## A.7.4 Task Scheduler ■■

Schedule tasks with cooldown using max heap.

### Code Example:

```
func leastInterval(_ tasks: [Character], _ n: Int) -> Int {
    var freq: [Character: Int] = [:]
    for task in tasks {
        freq[task, default: 0] += 1
    }

    var maxHeap = freq.values.sorted(by: >)
    var time = 0

    while !maxHeap.isEmpty {
        var temp: [Int] = []

        for i in 0...n {
            if !maxHeap.isEmpty {
                temp.append(maxHeap.removeFirst())
            }
        }

        for j in 0..<temp.count {
            temp[j] -= 1
            if temp[j] > 0 {
                maxHeap.append(temp[j])
            }
        }

        maxHeap.sort(by: >)
        time += maxHeap.isEmpty ? temp.count : n + 1
    }

    return time
}
```

### Key Points:

- **Time: O(n log m)**
- **Space: O(m)**
- **Max heap with cooldown**

### Notes:

*Use max heap to schedule most frequent tasks first.*

## A.7.5 Top K Frequent Elements ■■

Find k most frequent elements using heap.

## Code Example:

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

    // Use min heap of size k
    var minHeap: [(freq: Int, num: Int)] = []

    for (num, frequency) in freq {
        if minHeap.count < k {
            minHeap.append((frequency, num))
            minHeap.sort { $0.freq < $1.freq }
        } else if frequency > minHeap[0].freq {
            minHeap[0] = (frequency, num)
            minHeap.sort { $0.freq < $1.freq }
        }
    }

    return minHeap.map { $0.num }
}

// Alternative: Bucket sort approach
func topKFrequentBucket(_ nums: [Int], _ k: Int) -> [Int] {
    var freq: [Int: Int] = [:]
    for num in nums {
        freq[num, default: 0] += 1
    }

    var buckets: [[Int]] = Array(repeating: [], count: nums.count + 1)
    for (num, frequency) in freq {
        buckets[frequency].append(num)
    }

    var result: [Int] = []
    for i in stride(from: buckets.count - 1, through: 0, by: -1) {
        result.append(contentsOf: buckets[i])
        if result.count >= k { break }
    }

    return Array(result.prefix(k))
}
```

## Key Points:

- **Time: O(n log k) heap, O(n) bucket sort**
- **Space: O(n)**
- **Min heap or bucket sort**

## Notes:

*Use min heap of size k, or bucket sort for O(n) solution.*

## A.7.6 Find Median from Data Stream ■■■

Maintain running median using two heaps.

### Code Example:

```
class MedianFinder {
    private var maxHeap: [Int] = []  // Left half (max heap)
    private var minHeap: [Int] = []  // Right half (min heap)

    init() {}

    func addNum(_ num: Int) {
        // Add to max heap first
        maxHeap.append(num)
        maxHeap.sort(by: >)

        // Move largest from max heap to min heap
        if !maxHeap.isEmpty {
            minHeap.append(maxHeap.removeFirst())
            minHeap.sort()
        }

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

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

### Key Points:

- **Time: O(log n) addNum, O(1) findMedian**
- **Space: O(n)**
- **Two heaps technique**

### Notes:

*Use max heap for left half, min heap for right half.*

## A.7.7 Merge k Sorted Lists ■■■

Merge k sorted linked lists using min heap.

### Code Example:

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

    // Priority queue simulation with array
    var heap: [ListNode] = []

    // Add first node from each non-empty list
    for list in lists {
        if let node = list {
            heap.append(node)
        }
    }

    heap.sort { $0.val < $1.val }

    let dummy = ListNode(0)
    var current = dummy

    while !heap.isEmpty {
        let minNode = heap.removeFirst()
        current.next = minNode
        current = current.next!

        if let nextNode = minNode.next {
            heap.append(nextNode)
            heap.sort { $0.val < $1.val }
        }
    }

    return dummy.next
}
```

### Key Points:

**• Time: O(n log k)**
**• Space: O(k)**
**• Min heap with k nodes**

### Notes:

*Use min heap to always get smallest node among k lists.*

## A.7.8 Meeting Rooms II ■■

Find minimum meeting rooms using min heap for end times.

## Code Example:

```swift
func minMeetingRooms(_ intervals: [[Int]]) -> Int {
    let sortedIntervals = intervals.sorted { $0[0] < $1[0] }
    var endTimes: [Int] = []  // Min heap for end times

    for interval in sortedIntervals {
        let start = interval[0]
        let end = interval[1]

        // Remove meetings that have ended
        endTimes = endTimes.filter { $0 > start }

        // Add current meeting's end time
        endTimes.append(end)
        endTimes.sort()  // Maintain min heap property
    }

    return endTimes.count
}

// Alternative: Event-based approach
func minMeetingRoomsEvents(_ intervals: [[Int]]) -> Int {
    var events: [(time: Int, type: Int)] = []  // type: 1 for start, -1 for end

    for interval in intervals {
        events.append((interval[0], 1))   // meeting starts
        events.append((interval[1], -1))  // meeting ends
    }

    events.sort {
        if $0.time == $1.time {
            return $0.type < $1.type  // End before start
        }
        return $0.time < $1.time
    }

    var currentRooms = 0
    var maxRooms = 0

    for event in events {
        currentRooms += event.type
        maxRooms = max(maxRooms, currentRooms)
    }

    return maxRooms
}
```

## Key Points:

- **Time: O(n log n)**
- **Space: O(n)**
- **Min heap or event sorting**

## Notes:

*Track meeting end times with min heap, or use event-based sweep line.*

## A.8 Stack Problems (6 Problems)

### A.8.1 Valid Parentheses ■

Determine if parentheses string is valid.

#### Code Example:

```
func isValid(_ s: String) -> Bool {
    var stack: [Character] = []
    let pairs: [Character: Character] = [")": "(", "}": "{", "]": "["]

    for char in s {
        if let openBracket = pairs[char] {
            if stack.isEmpty || stack.removeLast() != openBracket {
                return false
            }
        } else {
            stack.append(char)
        }
    }

    return stack.isEmpty
}
```

#### Key Points:

- **Time: O(n)**
- **Space: O(n)**
- **Stack matching**

#### Notes:

*Use stack to match opening/closing brackets.*

### A.8.2 Min Stack ■■

Design stack with O(1) min operation.

#### Code Example:

```
class MinStack {
    private var stack: [Int] = []
    private var minStack: [Int] = []

    init() {}
    func push(_ val: Int) {
        stack.append(val)

        if minStack.isEmpty || val <= minStack.last! {
            minStack.append(val)
        }
    }

    func pop() {
        if let popped = stack.popLast() {
            if popped == minStack.last {
                minStack.removeLast()
            }
        }
    }

    func top() -> Int {
        return stack.last!
    }

    func getMin() -> Int {
        return minStack.last!
    }
}
```

### Key Points:

- **Time: O(1) all operations**
- **Space: O(n)**
- **Auxiliary stack**

### Notes:

*Use auxiliary stack to track minimum values.*

## A.8.3 Evaluate Reverse Polish Notation ■■

Evaluate expression in Reverse Polish Notation.

### Code Example:

```
func evalRPN(_ tokens: [String]) -> Int {
    var stack: [Int] = []

    for token in tokens {
        switch token {
        case "+":
            let b = stack.removeLast()
            let a = stack.removeLast()
            stack.append(a + b)
        case "-":
            let b = stack.removeLast()
            let a = stack.removeLast()
            stack.append(a - b)
        case "*":
            let b = stack.removeLast()
            let a = stack.removeLast()
            stack.append(a * b)
        case "/":
            let b = stack.removeLast()
            let a = stack.removeLast()
            stack.append(a / b)
        default:
            stack.append(Int(token)!)
        }
    }

    return stack.first!
}
```

### Key Points:

- **Time: O(n)**
- **Space: O(n)**
- **Stack evaluation**

### Notes:

*Use stack to evaluate postfix expressions.*

## A.8.4 Daily Temperatures ■■

Find next warmer temperature for each day.

### Code Example:

```
func dailyTemperatures(_ temperatures: [Int]) -> [Int] {
    var result = Array(repeating: 0, count: temperatures.count)
    var stack: [Int] = [] // indices

    for i in 0..<temperatures.count {
        while !stack.isEmpty && temperatures[i] > temperatures[stack.last!] {
            let prevIndex = stack.removeLast()
            result[prevIndex] = i - prevIndex
        }
        stack.append(i)
    }

    return result
}
```

### Key Points:

- **Time: O(n)**
- **Space: O(n)**
- **Monotonic decreasing stack**

### Notes:

*Use monotonic stack to find next greater element.*

## A.8.5 Car Fleet ■■

Count number of car fleets reaching destination.

### Code Example:

```
func carFleet(_ target: Int, _ position: [Int], _ speed: [Int]) -> Int {
    let cars = zip(position, speed).sorted { $0.0 > $1.0 }
    var stack: [Double] = []

    for (pos, spd) in cars {
        let timeToReach = Double(target - pos) / Double(spd)

        if stack.isEmpty || timeToReach > stack.last! {
            stack.append(timeToReach)
        }
    }

    return stack.count
}
```

### Key Points:

**• Time: O(n log n)**
**• Space: O(n)**
**• Stack simulation**

### Notes:

*Sort by position, use stack to track fleet formation.*

## A.8.6 Largest Rectangle in Histogram ■■■

Find area of largest rectangle in histogram.

### Code Example:

```
func largestRectangleArea(_ heights: [Int]) -> Int {
    var stack: [Int] = []
    var maxArea = 0
    var heights = heights + [0] // append 0 to process remaining bars

    for i in 0..<heights.count {
        while !stack.isEmpty && heights[i] < heights[stack.last!] {
            let height = heights[stack.removeLast()]
            let width = stack.isEmpty ? i : i - stack.last! - 1
            maxArea = max(maxArea, height * width)
        }
        stack.append(i)
    }

    return maxArea
}
```

## Key Points:

- **Time: O(n)**
- **Space: O(n)**
- **Monotonic increasing stack**

## Notes:

*Use stack to track indices, calculate areas when heights decrease.*

## A.9 Binary Search Problems (8 Problems)

### A.9.1 Binary Search ■

Search for target in sorted array.

#### Code Example:

```
func search(_ nums: [Int], _ target: Int) -> Int {
    var left = 0
    var right = nums.count - 1

    while left <= right {
        let mid = left + (right - left) / 2

        if nums[mid] == target {
            return mid
        } else if nums[mid] < target {
            left = mid + 1
        } else {
            right = mid - 1
        }
    }

    return -1
}
```

#### Key Points:

- **Time: O(log n)**
- **Space: O(1)**
- **Binary search**

#### Notes:

*Classic binary search with left/right pointers.*

### A.9.2 Search Insert Position ■

Find position where target should be inserted.

#### Code Example:

```
func searchInsert(_ nums: [Int], _ target: Int) -> Int {
    var left = 0
    var right = nums.count - 1
```

```
        while left <= right {
            let mid = left + (right - left) / 2

            if nums[mid] == target {
                return mid
            } else if nums[mid] < target {
                left = mid + 1
            } else {
                right = mid - 1
            }
        }

        return left
    }
```

### Key Points:

**• Time: O(log n)**
**• Space: O(1)**
**• Binary search variant**

### Notes:

*Same as binary search, return left pointer when not found.*

## A.9.3 Search in Rotated Sorted Array ■■

Search in rotated sorted array.

### Code Example:

```
    func search(_ nums: [Int], _ target: Int) -> Int {
        var left = 0
        var right = nums.count - 1

        while left <= right {
            let mid = left + (right - left) / 2

            if nums[mid] == target { return mid }

            if nums[left] <= nums[mid] {
                if nums[left] <= target && target < nums[mid] {
                    right = mid - 1
                } else {
                    left = mid + 1
                }
            } else {
                if nums[mid] < target && target <= nums[right] {
                    left = mid + 1
                } else {
                    right = mid - 1
                }
            }
        }

        return -1
    }
```

### Key Points:

**• Time: O(log n)**
**• Space: O(1)**
**• Modified binary search**

### Notes:

*Determine which half is sorted, then search appropriately.*

## A.9.4 Find Minimum in Rotated Array ■■

Find minimum element in rotated sorted array.

### Code Example:

```
func findMin(_ nums: [Int]) -> Int {
    var left = 0
    var right = nums.count - 1

    while left < right {
        let mid = left + (right - left) / 2

        if nums[mid] > nums[right] {
            left = mid + 1
        } else {
            right = mid
        }
    }

    return nums[left]
}
```

### Key Points:

**• Time: O(log n)**
**• Space: O(1)**
**• Binary search on rotation point**

### Notes:

*Find the pivot point where array was rotated.*

## A.9.5 Time Based Key-Value Store ■■

Get value at specific timestamp using binary search.

### Code Example:

```
class TimeMap {
    private var store: [String: [(String, Int)]] = [:]

    init() {}

    func set(_ key: String, _ value: String, _ timestamp: Int) {
        store[key, default: []].append((value, timestamp))
    }

    func get(_ key: String, _ timestamp: Int) -> String {
        guard let values = store[key] else { return "" }

        var left = 0, right = values.count - 1
        var result = ""

        while left <= right {
            let mid = left + (right - left) / 2

            if values[mid].1 <= timestamp {
                result = values[mid].0
                left = mid + 1
            } else {
                right = mid - 1
            }
        }

        return result
    }
}
```

### Key Points:

- **Time: O(log n) get, O(1) set**
- **Space: O(n)**
- **Binary search on timestamps**

### Notes:

*Store values with timestamps, binary search for closest past timestamp.*

## A.9.6 Search 2D Matrix ■■

Search target in row and column sorted matrix.

### Code Example:

```
func searchMatrix(_ matrix: [[Int]], _ target: Int) -> Bool {
    let m = matrix.count
    let n = matrix[0].count
    var left = 0
    var right = m * n - 1

    while left <= right {
        let mid = left + (right - left) / 2
        let midValue = matrix[mid / n][mid % n]

        if midValue == target {
            return true
        } else if midValue < target {
            left = mid + 1
        } else {
            right = mid - 1
        }
    }

    return false
}
```

### Key Points:

- **Time: O(log(m*n))**
- **Space: O(1)**
- **Treat 2D as 1D array**

### Notes:

*Map 2D coordinates to 1D index for binary search.*

## A.9.7 Koko Eating Bananas ■■

Find minimum eating speed to finish bananas in time.

### Code Example:

```
func minEatingSpeed(_ piles: [Int], _ h: Int) -> Int {
    var left = 1
    var right = piles.max()!

    func canFinish(_ speed: Int) -> Bool {
        var hours = 0
        for pile in piles {
            hours += (pile + speed - 1) / speed // ceiling division
        }
        return hours <= h
    }

    while left < right {
        let mid = left + (right - left) / 2

        if canFinish(mid) {
            right = mid
        } else {
            left = mid + 1
        }
    }

    return left
}
```

## Key Points:

- **Time: O(n * log(max))**
- **Space: O(1)**
- **Binary search on answer**

## Notes:

*Binary search on eating speed, check if feasible.*

## A.9.8 Median of Two Sorted Arrays ■■■

Find median of two sorted arrays in log time.

## Code Example:

```
func findMedianSortedArrays(_ nums1: [Int], _ nums2: [Int]) -> Double {
    let (a, b) = nums1.count <= nums2.count ? (nums1, nums2) : (nums2, nums1)
    let m = a.count, n = b.count
    let half = (m + n + 1) / 2

    var left = 0, right = m

    while left <= right {
        let i = (left + right) / 2
        let j = half - i

        let maxLeftA = i == 0 ? Int.min : a[i - 1]
        let minRightA = i == m ? Int.max : a[i]
        let maxLeftB = j == 0 ? Int.min : b[j - 1]
        let minRightB = j == n ? Int.max : b[j]

        if maxLeftA <= minRightB && maxLeftB <= minRightA {
            if (m + n) % 2 == 0 {
                return Double(max(maxLeftA, maxLeftB) + min(minRightA, minRightB)) / 2.0
            } else {
                return Double(max(maxLeftA, maxLeftB))
            }
        } else if maxLeftA > minRightB {
            right = i - 1
        } else {
            left = i + 1
        }
    }
```

```
        return 0.0
    }
```

## Key Points:

- **Time: O(log(min(m,n)))**
- **Space: O(1)**
- **Binary search partition**

## Notes:

*Binary search on partition point to find median.*

■ SUMMARY: This appendix contains 100+ essential DSA problems with optimized Swift solutions. Each problem is carefully selected for technical interviews and includes multiple solution approaches where applicable. The problems progress from basic to advanced, covering all major algorithmic patterns and data structures. ■ = Easy, ■■ = Medium, ■■■ = Hard Total Problems: 111 problems across 9 categories • Arrays: 20 problems • Strings: 15 problems • Linked Lists: 12 problems • Binary Trees: 15 problems • Dynamic Programming: 15 problems • Graphs: 12 problems • Heaps: 8 problems • Stacks: 6 problems • Binary Search: 8 problems