# Data Structures & Algorithms Problems and Solutions

**Complete Swift Implementation Guide**

Generated on: September 11, 2025

# Table of Contents

# Array Problems

## Array: Two Sum

### *Problem Description:*

Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to target. You may assume that each input would have exactly one solution, and you may not use the same element twice.

### *Swift Solution:*

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

    for (index, num) in nums.enumerated() {
        let complement = target - num

        if let complementIndex = numToIndex[complement] {
            return [complementIndex, index]
        }

        numToIndex[num] = index
    }

    return [] // No solution found
}

// Example usage:
let nums = [2, 7, 11, 15]
let target = 9
let result = twoSum(nums, target)
print(result) // Output: [0, 1]
```

### *Explanation:*

We use a hash map to store each number and its index as we iterate through the array. For each number, we calculate its complement (target - current number) and check if it exists in our hash map. If found, we return the indices.

### *Complexity Analysis:*

**Time Complexity: O(n)**
**Space Complexity: O(n)**

## Array: Maximum Subarray (Kadane's Algorithm)

## Problem Description:

Given an integer array nums, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum.

## Swift Solution:

```
func maxSubArray(_ nums: [Int]) -> Int {
    guard !nums.isEmpty else { return 0 }

    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
}

// Example usage:
let nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
let result = maxSubArray(nums)
print(result) // Output: 6 (subarray [4, -1, 2, 1])
```

## Explanation:

Kadane's algorithm maintains the maximum sum ending at each position. At each step, we decide whether to extend the existing subarray or start a new one from the current element.

## Complexity Analysis:

**Time Complexity: O(n)**
**Space Complexity: O(1)**

# Array: Merge Sorted Arrays

## Problem Description:

You are given two integer arrays nums1 and nums2, sorted in non-decreasing order, and two integers m and n, representing the number of elements in nums1 and nums2 respectively. Merge nums1 and nums2 into a single array sorted in non-decreasing order.

## Swift Solution:

```
func merge(_ nums1: inout [Int], _ m: Int, _ nums2: [Int], _ n: Int) {
    var i = m - 1  // Last element in nums1
    var j = n - 1  // Last element in nums2
    var k = m + n - 1  // Last position in nums1

    // Merge from the end to avoid overwriting
```

```
    while i >= 0 && j >= 0 {
        if nums1[i] > nums2[j] {
            nums1[k] = nums1[i]
            i -= 1
        } else {
            nums1[k] = nums2[j]
            j -= 1
        }
        k -= 1
    }

    // Copy remaining elements from nums2
    while j >= 0 {
        nums1[k] = nums2[j]
        j -= 1
        k -= 1
    }
}

// Example usage:
var nums1 = [1, 2, 3, 0, 0, 0]
let nums2 = [2, 5, 6]
merge(&nums1, 3, nums2, 3)
print(nums1) // Output: [1, 2, 2, 3, 5, 6]
```

## *Explanation:*

We merge from the end of both arrays to avoid overwriting elements in nums1. This allows us to merge in-place without extra space.

## *Complexity Analysis:*

**Time Complexity: O(m + n)**
**Space Complexity: O(1)**

# Linked List Problems

## Linked List: Reverse Linked List

### *Problem Description:*

Given the head of a singly linked list, reverse the list, and return the reversed list.

### *Swift Solution:*

```
class ListNode {
    var val: Int
    var next: ListNode?

    init(_ val: Int) {
        self.val = val
        self.next = nil
    }
}

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

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

    return prev
}

// Recursive approach:
func reverseListRecursive(_ head: ListNode?) -> ListNode? {
    guard let head = head, let next = head.next else {
        return head
    }

    let reversedList = reverseListRecursive(next)
    next.next = head
    head.next = nil

    return reversedList
}
```

### *Explanation:*

The iterative approach uses three pointers to reverse the links. The recursive approach reverses the rest of the list first, then fixes the current connection.

### *Complexity Analysis:*

**Time Complexity: O(n)**
**Space Complexity: O(1) iterative, O(n) recursive**

## Linked List: Merge Two Sorted Lists

### *Problem Description:*

You are given the heads of two sorted linked lists list1 and list2. Merge the two lists in a one sorted list. The list should be made by splicing together the nodes of the first two lists.

### *Swift Solution:*

```
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!
    }

    // Append remaining nodes
    current.next = l1 ?? l2

    return dummy.next
}

// Example usage:
// list1: 1 -> 2 -> 4
// list2: 1 -> 3 -> 4
// result: 1 -> 1 -> 2 -> 3 -> 4 -> 4
```

### *Explanation:*

We use a dummy node to simplify the logic and iterate through both lists, always choosing the smaller value. Finally, we append any remaining nodes.

### *Complexity Analysis:*

**Time Complexity: O(m + n)**
**Space Complexity: O(1)**

# Linked List: Detect Cycle

## *Problem Description:*

Given head, the head of a linked list, determine if the linked list has a cycle in it. There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer.

## *Swift Solution:*

```
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
}

// Finding the start of the cycle
func detectCycle(_ head: ListNode?) -> ListNode? {
    var slow = head
    var fast = head

    // First, detect if there's a cycle
    while fast != nil && fast?.next != nil {
        slow = slow?.next
        fast = fast?.next?.next

        if slow === fast {
            break
        }
    }

    // No cycle found
    if fast == nil || fast?.next == nil {
        return nil
    }

    // Find the start of the cycle
    slow = head
    while slow !== fast {
        slow = slow?.next
```

```
            fast = fast?.next
        }

        return slow
    }
```

## *Explanation:*

Floyd's Cycle Detection Algorithm (Tortoise and Hare): Use two pointers moving at different speeds. If there's a cycle, the fast pointer will eventually meet the slow pointer.

## *Complexity Analysis:*

**Time Complexity: O(n)**
**Space Complexity: O(1)**

# Binary Tree Problems

## Binary Tree: Binary Tree Inorder Traversal

### *Problem Description:*

Given the root of a binary tree, return the inorder traversal of its nodes' values. (Left, Root, Right)

### *Swift Solution:*

```
class TreeNode {
    var val: Int
    var left: TreeNode?
    var right: TreeNode?

    init(_ val: Int) {
        self.val = val
        self.left = nil
        self.right = nil
    }
}

// Inorder Traversal
func inorderTraversal(_ root: TreeNode?) -> [Int] {
    var result: [Int] = []

    func inorder(_ node: TreeNode?) {
        guard let node = node else { return }

        inorder(node.left)
        result.append(node.val)
        inorder(node.right)
    }

    inorder(root)
    return result
}

// Iterative approach using stack
func inorderTraversalIterative(_ root: TreeNode?) -> [Int] {
    var result: [Int] = []
    var stack: [TreeNode] = []
    var current = root

    while current != nil || !stack.isEmpty {
        while current != nil {
            stack.append(current!)
            current = current!.left
        }
```

```
            current = stack.removeLast()
            result.append(current!.val)
            current = current!.right
        }

        return result
    }
```

### *Explanation:*

Inorder traversal visits left subtree, then root, then right subtree. The recursive approach is natural, while the iterative approach uses a stack to simulate the recursion.

### *Complexity Analysis:*

**Time Complexity: O(n)**
**Space Complexity: O(h) where h is height of tree**

## Binary Tree: Maximum Depth of Binary Tree

### *Problem Description:*

Given the root of a binary tree, return its maximum depth. A binary tree's maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

### *Swift Solution:*

```
func maxDepth(_ root: TreeNode?) -> Int {
    guard let root = root else { return 0 }

    let leftDepth = maxDepth(root.left)
    let rightDepth = maxDepth(root.right)

    return max(leftDepth, rightDepth) + 1
}

// Iterative approach using level-order traversal
func maxDepthIterative(_ root: TreeNode?) -> Int {
    guard let root = root else { return 0 }

    var queue: [TreeNode] = [root]
    var depth = 0

    while !queue.isEmpty {
        let levelSize = queue.count
        depth += 1

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

            if let left = node.left {
                queue.append(left)
```

```swift
            }
            if let right = node.right {
                queue.append(right)
            }
        }
    }

    return depth
}
```

### *Explanation:*

The recursive approach calculates depth by finding the maximum of left and right subtree depths plus 1. The iterative approach uses level-order traversal to count levels.

### *Complexity Analysis:*

**Time Complexity: O(n)**
**Space Complexity: O(h) recursive, O(w) iterative where h is height and w is width**

## Binary Tree: Validate Binary Search Tree

### *Problem Description:*

Given the root of a binary tree, determine if it is a valid binary search tree (BST). A valid BST is defined as follows: The left subtree of a node contains only nodes with keys less than the node's key. The right subtree of a node contains only nodes with keys greater than the node's key. Both the left and right subtrees must also be binary search trees.

### *Swift Solution:*

```swift
func isValidBST(_ root: TreeNode?) -> Bool {
    return validate(root, nil, nil)
}

func validate(_ node: TreeNode?, _ minVal: Int?, _ maxVal: Int?) -> Bool {
    guard let node = node else { return true }

    // Check if current node violates BST property
    if let minVal = minVal, node.val <= minVal { return false }
    if let maxVal = maxVal, node.val >= maxVal { return false }

    // Recursively validate left and right subtrees
    return validate(node.left, minVal, node.val) &&
           validate(node.right, node.val, maxVal)
}

// Alternative approach using inorder traversal
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)
}
```

## *Explanation:*

We can validate by maintaining min/max bounds for each node, or by doing inorder traversal and checking if the result is sorted.

## *Complexity Analysis:*

**Time Complexity: O(n)**
**Space Complexity: O(h) where h is the height of the tree**

# Stack and Queue Problems

## Stack: Valid Parentheses

### *Problem Description:*

Given a string s containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid. An input string is valid if: Open brackets must be closed by the same type of brackets. Open brackets must be closed in the correct order.

### *Swift Solution:*

```
func isValid(_ s: String) -> Bool {
    var stack: [Character] = []
    let pairs: [Character: Character] = [")": "(", "}": "{", "]": "["]

    for char in s {
        if pairs.keys.contains(char) {
            // Closing bracket
            if stack.isEmpty || stack.removeLast() != pairs[char] {
                return false
            }
        } else {
            // Opening bracket
            stack.append(char)
        }
    }

    return stack.isEmpty
}

// Example usage:
print(isValid("()"))        // true
print(isValid("()[]{}"))    // true
print(isValid("(]"))        // false
print(isValid("([)]"))      // false
```

### *Explanation:*

Use a stack to keep track of opening brackets. When we encounter a closing bracket, check if it matches the most recent opening bracket.

### *Complexity Analysis:*

**Time Complexity: O(n)**
**Space Complexity: O(n)**

# Queue: Implement Queue using Stacks

## *Problem Description:*

Implement a first in first out (FIFO) queue using only two stacks. The implemented queue should support all the functions of a normal queue (push, peek, pop, and empty).

## *Swift Solution:*

```
class MyQueue {
    private var inStack: [Int] = []
    private var outStack: [Int] = []

    init() {}

    func push(_ x: Int) {
        inStack.append(x)
    }

    func pop() -> Int {
        peek()
        return outStack.removeLast()
    }

    func peek() -> Int {
        if outStack.isEmpty {
            while !inStack.isEmpty {
                outStack.append(inStack.removeLast())
            }
        }
        return outStack.last!
    }

    func empty() -> Bool {
        return inStack.isEmpty && outStack.isEmpty
    }
}

// Example usage:
let queue = MyQueue()
queue.push(1)
queue.push(2)
print(queue.peek())  // 1
print(queue.pop())   // 1
print(queue.empty()) // false
```

## *Explanation:*

Use two stacks: one for input and one for output. Transfer elements from input to output stack only when output stack is empty.

## *Complexity Analysis:*

**Time Complexity: O(1) amortized for all operations**
**Space Complexity: O(n)**

# Graph Problems

## Graph: Breadth-First Search (BFS)

### *Problem Description:*

Implement breadth-first search traversal for a graph. BFS explores all vertices at the present depth prior to moving on to vertices at the next depth level.

### *Swift Solution:*

```
// Graph represented as adjacency list
func bfs(_ graph: [Int: [Int]], start: Int) -> [Int] {
    var visited: Set<Int> = []
    var queue: [Int] = [start]
    var result: [Int] = []

    visited.insert(start)

    while !queue.isEmpty {
        let node = queue.removeFirst()
        result.append(node)

        if let neighbors = graph[node] {
            for neighbor in neighbors {
                if !visited.contains(neighbor) {
                    visited.insert(neighbor)
                    queue.append(neighbor)
                }
            }
        }
    }

    return result
}

// Example usage:
let graph = [
    0: [1, 2],
    1: [2],
    2: [0, 3],
    3: [3]
]
let bfsResult = bfs(graph, start: 2)
print(bfsResult) // [2, 0, 3, 1]
```

### *Explanation:*

BFS uses a queue to process nodes level by level. We mark nodes as visited to avoid cycles and process all neighbors before moving to the next level.

## *Complexity Analysis:*

**Time Complexity: O(V + E)**
**Space Complexity: O(V)**

# Graph: Depth-First Search (DFS)

## *Problem Description:*

Implement depth-first search traversal for a graph. DFS explores as far as possible along each branch before backtracking.

## *Swift Solution:*

```
func dfs(_ graph: [Int: [Int]], start: Int) -> [Int] {
    var visited: Set<Int> = []
    var result: [Int] = []

    func dfsHelper(_ node: Int) {
        visited.insert(node)
        result.append(node)

        if let neighbors = graph[node] {
            for neighbor in neighbors {
                if !visited.contains(neighbor) {
                    dfsHelper(neighbor)
                }
            }
        }
    }

    dfsHelper(start)
    return result
}

// Iterative DFS using stack
func dfsIterative(_ graph: [Int: [Int]], start: Int) -> [Int] {
    var visited: Set<Int> = []
    var stack: [Int] = [start]
    var result: [Int] = []

    while !stack.isEmpty {
        let node = stack.removeLast()

        if !visited.contains(node) {
            visited.insert(node)
            result.append(node)

            if let neighbors = graph[node] {
                for neighbor in neighbors.reversed() {
                    if !visited.contains(neighbor) {
```

```
                        stack.append(neighbor)
                    }
                }
            }
        }
    }

    return result
}
```

### *Explanation:*

DFS can be implemented recursively or iteratively using a stack. We explore each path completely before backtracking to explore other paths.

### *Complexity Analysis:*

**Time Complexity: O(V + E)**
**Space Complexity: O(V)**

## Graph: Shortest Path (Unweighted)

### *Problem Description:*

Find the shortest path between two nodes in an unweighted graph. Return the path as a list of nodes from start to end.

### *Swift Solution:*

```
// Simplified shortest path using BFS for unweighted graphs
func shortestPath(_ graph: [Int: [Int]], start: Int, end: Int) -> [Int]? {
    var queue: [(Int, [Int])] = [(start, [start])]
    var visited: Set<Int> = [start]

    while !queue.isEmpty {
        let (node, path) = queue.removeFirst()

        if node == end {
            return path
        }

        if let neighbors = graph[node] {
            for neighbor in neighbors {
                if !visited.contains(neighbor) {
                    visited.insert(neighbor)
                    queue.append((neighbor, path + [neighbor]))
                }
            }
        }
    }

    return nil // No path found
```

```
}

// Example usage:
let graph = [
    0: [1, 2],
    1: [2, 3],
    2: [3],
    3: []
]
if let path = shortestPath(graph, start: 0, end: 3) {
    print("Shortest path: \(path)") // [0, 1, 3] or [0, 2, 3]
}
```

## *Explanation:*

For unweighted graphs, BFS naturally finds the shortest path since it explores nodes level by level, guaranteeing the first path found is the shortest.

## *Complexity Analysis:*

**Time Complexity: O(V + E)**
**Space Complexity: O(V)**