Here is a structured presentation of solutions to the 15 Swift interview questions, each with an explanation, code, Q&A, basic lessons, and complexity analysis:

1. FizzBuzz

Problem: Print numbers from 1 to n, replacing multiples of 3 with "Fizz", multiples of 5 with "Buzz", and both with "FizzBuzz".

Solution:

```swift
func fizzbuzz(n: Int) -> [String] {
    var result = [String]()
    for i in 1...n {
        if i % 15 == 0 {
            result.append("FizzBuzz")
        } else if i % 3 == 0 {
            result.append("Fizz")
        } else if i % 5 == 0 {
            result.append("Buzz")
        } else {
            result.append(String(i))
        }
    }
    return result
}
```

Q&A: Why check for 15 first? To avoid multiple checks and ensure correct order.

Lesson: Use modulo operations efficiently to minimize checks.

Complexity: O(n)

2. Reverse String

Problem: Reverse a given string.

Solution:

```swift
func reverseString(s: String) -> String {
    let chars = Array(s)
    return String(chars.reversed())
}
```

Q&A: How handle Unicode? Swift's `reversed()` manages extended grapheme clusters.

Lesson: Leverage built-in functions for simplicity and efficiency.

Complexity: O(n)

3. Unique Characters in String

Problem: Check if a string has all unique characters.

Solution:

```
func hasUniqueCharacters(s: String) -> Bool {
    var seen = Set<Character>()
    for char in s {
        if seen.contains(char) {
            return false
        }
        seen.insert(char)
    }
    return true
}
```

Q&A: Does it handle Unicode? Yes, using `Character` type.

Lesson: Use hash sets for quick lookups.

Complexity: O(n)

4. Reverse Complement of DNA String

Problem: Reverse and complement a DNA string (A<->T, C<->G).

Solution:

```
func reverseComplement(s: String) -> String? {
    let mapping = ["A": "T", "T": "A", "C": "G", "G": "C"]
    var result = ""
    for char in s.reversed() {
        if let mapped = mapping[String(char)] {
            result += mapped
        } else {
            return nil // Invalid character
        }
    }
    return String(result)
}
```

Q&A: How handle invalid characters? Return nil or throw an error.

Lesson: Use dictionaries for quick lookups and iterate in reverse.

Complexity: O(n)

5. Maximum Single-Sell Profit

Problem: Find max profit from a stock price array.

Solution:

```
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
}
```

Q&A: What if no profit? Returns 0.

Lesson: Track minimum and update profits iteratively.

Complexity: O(n)

6. First Non-Repeating Character

Problem: Find the first character in a string that appears once.

Solution:

```
func firstNonRepeatingCharacter(s: String) -> Character? {
    var frequency = [Character: Int]()
    for char in s {
        frequency[char] = (frequency[char] ?? 0) + 1
    }
    for char in s {
        if frequency[char] == 1 {
            return char
        }
    }
    return nil
}
```

Q&A: How handle empty string? Returns nil.

Lesson: Two-pass approach ensures efficiency.

Complexity: O(n)

7. Check Anagrams

Problem: Determine if two strings are anagrams.

Solution:

```
func areAnagrams(_ s1: String, _ s2: String) -> Bool {
    return String(s1.sorted()) == String(s2.sorted())
}
```

Q&A: Why sort? To easily compare character sequences.

Lesson: Sorting is a straightforward method for anagrams.

Complexity: O(n log n)

8. Valid Parentheses

Problem: Check if parentheses string is valid.

Solution:

```
func isValidParentheses(s: String) -> Bool {
    var stack = [Character]()
    let mapping = ["(": ")", "[" : "]", "{" : "}"]
    for char in s {
        if let value = mapping[char] {
            stack.append(value)
        } else if char == ")" || char == "]" || char == "}" {
            if stack.isEmpty || stack.pop() != char {
                return false
            }
        }
    }
    return stack.isEmpty
}
```

Q&A: What about different types? Each closing must match last opening.

Lesson: Use stack to track expected closing parentheses.

Complexity: O(n)

9. Missing Number (1 to n)

Problem: Find missing number in an unsorted array of 1..n.

Solution:

```
func findMissingNumber(nums: [Int]) -> Int {
    let n = nums.count + 1 // since one is missing
    return n * (n + 1) / 2 - nums.reduce(0, +)
}
```

Q&A: Why this works? Sum of first n numbers minus actual sum gives missing.

Lesson: Mathematical approach for efficiency.

Complexity: O(n)

10. Valid Palindrome

Problem: Check if a string is a palindrome, ignoring non-alphanumeric and case.

Solution:

```
func isPalindrome(s: String) -> Bool {
    let filtered = s.lowercased().filter { $0.isLetter || $0.isNumber }
    var left = 0
    var right = filtered.count - 1
    while left < right {
        if filtered[filtered.index(filtered.startIndex, offsetBy: left)] !=
filtered[filtered.index(filtered.startIndex, offsetBy: right)] {
            return false
        }
        left += 1
        right -= 1
    }
    return true
}
```

Q&A: How handle spaces and cases? Ignore them during filtering.

Lesson: Use two pointers to compare from both ends.

Complexity: O(n)

11. Game of Life

Problem: Implement Conway's Game of Life.

Solution:

```
func gameOfLife(board: [[Bool]]) -> [[Bool]] {
    guard !board.isEmpty else { return board }
    let rows = board.count
    let cols = board[0].count
    var newBoard = board.map { $0.map { $0 } }

    for i in 0..<rows {
        for j in 0..<cols {
            let neighbors = getNeighbors(board, i, j)
            if !board[i][j] && neighbors == 3 {
                newBoard[i][j] = true
            } else if board[i][j] && (neighbors < 2 || neighbors > 3) {
                newBoard[i][j] = false
            }
```

```
        }
    }

    return newBoard
}
```

Q&A: How count neighbors? Check all eight surrounding cells.

Lesson: Use a helper function to count live neighbors.

Complexity: O(m*n)

12. String Contains Duplicate Characters

Problem: Determine if string has duplicate characters.

Solution:

```
func containsDuplicates(s: String) -> Bool {
    var seen = Set<Character>()
    for char in s {
        if seen.contains(char) {
            return true
        }
        seen.insert(char)
    }
    return false
}
```

Q&A: What about case sensitivity? Currently considers 'A' and 'a' different.

Lesson: Use a set to track encountered characters.

Complexity: O(n)

13. Implement Queue Using Two Stacks

Problem: Create queue using two stacks.

Solution:

```
class Queue<T> {
    private var stack1 = [T]()
    private var stack2 = [T]()

    func enqueue(_ element: T) {
        stack1.append(element)
    }

    func dequeue() -> T? {
        if stack2.isEmpty {
```

```
            while !stack1.isEmpty {
                stack2.append(stack1.pop()!)
            }
        }
        return stack2.pop()
    }
}
```

Q&A: How handle empty queue? Dequeue returns nil.

Lesson: Use two stacks to mimic FIFO behavior.

Complexity: Amortized O(1) per operation.

14. Roman Numeral Conversion

Problem: Convert integer to Roman numeral string.

Solution:

```
func intToRoman(_ num: Int) -> String {
    let values = [1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1]
    let symbols = ["M", "CM", "D", "CD", "C", "XC", "L", "XL", "X", "IX", "V", "IV", "I"]

    var roman = ""
    for (value, symbol) in zip(values, symbols) {
        while num >= value {
            roman += symbol
            num -= value
        }
    }
    return roman
}
```

Q&A: How handle numbers beyond 3999? Typically not required.

Lesson: Use a lookup table and subtract values iteratively.

Complexity: O(n)

15. Linked List Intersection

Problem: Find intersection node of two linked lists.

Solution:

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

```swift
func getIntersectionNode(headA: ListNode?, headB: ListNode?) -> ListNode? {
    guard let a = headA, let b = headB else { return nil }

    var p1 = a
    var p2 = b

    while p1 !== p2 {
        p1 = p1.next ?? b
        p2 = p2.next ?? a
    }

    return p1
}
```

Q&A: What if no intersection? Loops until meeting point.

Lesson: Use pointer manipulation to find common node efficiently.

Complexity: O(m + n)

Each problem is designed to test specific programming concepts and Swift language features, ensuring a comprehensive understanding for an interview setting.