Basic DSA Questions for beginners

1. ARRAYS

Question 1: Find Maximum Element in Array

Problem: Given an array of integers, find the maximum element.

Pseudocode:

```
FUNCTION findMax(array):
    IF array is empty:
        RETURN null

    max = array[0]
    FOR i = 1 to array.length - 1:
        IF array[i] > max:
            max = array[i]
    RETURN max
```

Explanation: We initialize max with the first element and traverse the array once, updating max whenever we find a larger element. Time complexity: O(n), Space complexity: O(1).

Question 2: Reverse an Array

Problem: Reverse the elements of an array in-place.

Pseudocode:

```
FUNCTION reverseArray(array):
    left = 0
    right = array.length - 1

    WHILE left < right:
        SWAP array[left] and array[right]
        left = left + 1
        right = right - 1
```

Explanation: Use two pointers approach - one from start, one from end. Swap elements and move pointers towards center until they meet. Time complexity: O(n), Space complexity: O(1).

Question 3: Find Second Largest Element

Problem: Find the second largest element in an array.

Pseudocode:

```
FUNCTION findSecondLargest(array):
    IF array.length < 2:
        RETURN null

    largest = -infinity
    secondLargest = -infinity

    FOR each element in array:
        IF element > largest:
            secondLargest = largest
            largest = element
        ELSE IF element > secondLargest AND element != largest:
            secondLargest = element

    RETURN secondLargest if secondLargest != -infinity else null
```

Explanation: Single pass algorithm that maintains two variables for largest and second largest. We update them appropriately as we traverse. Time complexity: O(n), Space complexity: O(1).

Question 4: Remove Duplicates from Sorted Array

Problem: Remove duplicates from a sorted array in-place.

Pseudocode:

```
FUNCTION removeDuplicates(sortedArray):
    IF array is empty:
        RETURN 0

    writeIndex = 1

    FOR readIndex = 1 to array.length - 1:
        IF array[readIndex] != array[readIndex - 1]:
            array[writeIndex] = array[readIndex]
            writeIndex = writeIndex + 1

    RETURN writeIndex
```

Explanation: Two pointer technique where readIndex scans the array and writeIndex marks position for next unique element. Since array is sorted, duplicates are adjacent. Time complexity: O(n), Space complexity: O(1).

Question 5: Move Zeros to End

Problem: Move all zeros in an array to the end while maintaining relative order of non-zero elements.

Pseudocode:

```

```

```
FUNCTION moveZerosToEnd(array):
    writeIndex = 0

    FOR readIndex = 0 to array.length - 1:
        IF array[readIndex] != 0:
            array[writeIndex] = array[readIndex]
            writeIndex = writeIndex + 1

    WHILE writeIndex < array.length:
        array[writeIndex] = 0
        writeIndex = writeIndex + 1
```

Explanation: First pass moves all non-zero elements to the front, second pass fills remaining positions with zeros. Maintains relative order of non-zero elements. Time complexity: O(n), Space complexity: O(1).

Question 6: Find Missing Number

Problem: Given an array containing n distinct numbers from 0 to n, find the missing number.

Pseudocode:

```
FUNCTION findMissingNumber(array, n):
    expectedSum = n * (n + 1) / 2
    actualSum = 0

    FOR each element in array:
        actualSum = actualSum + element

    RETURN expectedSum - actualSum
```

Explanation: Uses mathematical approach - sum of first n natural numbers minus actual sum gives missing number. Alternative: XOR all numbers from 0 to n with all array elements. Time complexity: O(n), Space complexity: O(1).

Question 7: Rotate Array by K Positions

Problem: Rotate array to the right by k steps.

Pseudocode:

```
FUNCTION rotateArray(array, k):
    n = array.length
    k = k % n  // Handle k > n

    // Reverse entire array
    REVERSE(array, 0, n - 1)

    // Reverse first k elements
```

```
    REVERSE(array, 0, k - 1)

    // Reverse remaining elements
    REVERSE(array, k, n - 1)

FUNCTION reverse(array, start, end):
    WHILE start < end:
        SWAP array[start] and array[end]
        start = start + 1
        end = end - 1
```

Explanation: Three-step reversal algorithm. Reverse entire array, then reverse first k elements, then reverse rest. Time complexity: O(n), Space complexity: O(1).

Question 8: Check if Array is Sorted

Problem: Check if an array is sorted in non-decreasing order.

Pseudocode:

```
FUNCTION isSorted(array):
    FOR i = 1 to array.length - 1:
        IF array[i] < array[i - 1]:
            RETURN false
    RETURN true
```

Explanation: Single pass through array checking if each element is greater than or equal to previous element. Time complexity: O(n), Space complexity: O(1).

Question 9: Find Intersection of Two Arrays

Problem: Find common elements between two arrays.

Pseudocode:

```
FUNCTION findIntersection(array1, array2):
    result = []

    FOR each element1 in array1:
        FOR each element2 in array2:
            IF element1 == element2:
                IF element1 not in result:
                    ADD element1 to result
                BREAK

    RETURN result
```

Explanation: Nested loop approach for basic solution. For each element in first array, check if it exists in second array. Can be optimized using hash sets. Time complexity: O(n*m), Space complexity: O(min(n,m)).

Question 10: Leaders in Array

Problem: Find all leaders in array (element is leader if all elements to its right are smaller).

Pseudocode:

```
FUNCTION findLeaders(array):
    n = array.length
    leaders = []
    maxFromRight = array[n - 1]

    ADD array[n - 1] to leaders  // Rightmost is always leader

    FOR i = n - 2 down to 0:
        IF array[i] > maxFromRight:
            ADD array[i] to leaders
            maxFromRight = array[i]

    REVERSE leaders array
    RETURN leaders
```

Explanation: Traverse from right to left, maintaining maximum element seen so far from right. If current element is greater than this maximum, it's a leader. Time complexity: O(n), Space complexity: O(1) extra.

---

2. STRINGS

Question 1: Check if String is Palindrome

Problem: Check if a given string reads the same forwards and backwards.

Pseudocode:

```
FUNCTION isPalindrome(string):
    left = 0
    right = string.length - 1

    WHILE left < right:
        IF string[left] != string[right]:
            RETURN false
        left = left + 1
        right = right - 1

    RETURN true
```

Explanation: Two pointer approach comparing characters from both ends moving towards center. If any pair doesn't match, it's not a palindrome. Time complexity: O(n), Space complexity: O(1).

Question 2: Reverse a String

Problem: Reverse the characters in a string.

Pseudocode:

```
FUNCTION reverseString(string):
    charArray = CONVERT string to character array
    left = 0
    right = charArray.length - 1

    WHILE left < right:
        SWAP charArray[left] and charArray[right]
        left = left + 1
        right = right - 1

    RETURN CONVERT charArray to string
```

Explanation: Convert string to character array for in-place manipulation, then use two pointers to swap characters from ends towards center. Time complexity: O(n), Space complexity: O(n) for character array.

Question 3: Check Anagrams

Problem: Check if two strings are anagrams of each other.

Pseudocode:

```
FUNCTION areAnagrams(string1, string2):
    IF string1.length != string2.length:
        RETURN false

    charCount = ARRAY of size 256 initialized to 0

    FOR i = 0 to string1.length - 1:
        charCount[ASCII(string1[i])] = charCount[ASCII(string1[i])] + 1
        charCount[ASCII(string2[i])] = charCount[ASCII(string2[i])] - 1

    FOR i = 0 to 255:
        IF charCount[i] != 0:
            RETURN false

    RETURN true
```

Explanation: Count frequency of each character in first string (increment) and second string (decrement). If strings are anagrams, all counts should be zero. Time complexity: O(n), Space complexity: O(1) - fixed size array.

Question 4: Find First Non-Repeating Character

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

Pseudocode:

```
FUNCTION firstNonRepeating(string):
    charCount = ARRAY of size 256 initialized to 0

    // Count frequency of each character
    FOR each character in string:
        charCount[ASCII(character)] = charCount[ASCII(character)] + 1

    // Find first character with count 1
    FOR each character in string:
        IF charCount[ASCII(character)] == 1:
            RETURN character

    RETURN null
```

Explanation: Two pass algorithm - first pass counts frequencies, second pass finds first character with count 1. Time complexity: O(n), Space complexity: O(1) - fixed size array.

Question 5: Remove Duplicates from String

Problem: Remove duplicate characters from string keeping first occurrence.

Pseudocode:

```
FUNCTION removeDuplicates(string):
    seen = BOOLEAN ARRAY of size 256 initialized to false
    result = ""

    FOR each character in string:
        IF seen[ASCII(character)] == false:
            result = result + character
            seen[ASCII(character)] = true

    RETURN result
```

Explanation: Use boolean array to track seen characters. Add character to result only if not seen before. Time complexity: O(n), Space complexity: O(1) for tracking + O(n) for result.

Question 6: Check if Strings are Rotation of Each Other

Problem: Check if one string is rotation of another (e.g., "abcde" and "cdeab").

Pseudocode:

```
FUNCTION areRotations(string1, string2):
    IF string1.length != string2.length:
        RETURN false

    concatenated = string1 + string1
    RETURN concatenated.contains(string2)
```

Explanation: If string2 is rotation of string1, then string2 will be substring of string1+string1. This works because concatenating string1 with itself contains all possible rotations as substrings. Time complexity: O(n), Space complexity: O(n).

Question 7: Longest Common Prefix

Problem: Find longest common prefix among array of strings.

Pseudocode:

```
FUNCTION longestCommonPrefix(strings):
    IF strings is empty:
        RETURN ""

    prefix = strings[0]

    FOR i = 1 to strings.length - 1:
        WHILE strings[i] does not start with prefix:
            prefix = prefix.substring(0, prefix.length - 1)
            IF prefix is empty:
                RETURN ""

    RETURN prefix
```

Explanation: Start with first string as prefix, then for each subsequent string, reduce prefix until it matches the beginning of current string. Time complexity: O(S) where S is sum of all string lengths, Space complexity: O(1).

Question 8: Count Vowels and Consonants

Problem: Count number of vowels and consonants in a string.

Pseudocode:

```
FUNCTION countVowelsConsonants(string):
    vowels = 0
    consonants = 0
    vowelSet = {'a', 'e', 'i', 'o', 'u', 'A', 'E', 'I', 'O', 'U'}

    FOR each character in string:
        IF character is alphabetic:
            IF character in vowelSet:
```

```
                vowels = vowels + 1
            ELSE:
                consonants = consonants + 1

    RETURN (vowels, consonants)
```

Explanation: Single pass through string, categorizing each alphabetic character as vowel or consonant using a set for quick lookup. Time complexity: O(n), Space complexity: O(1).

Question 9: Reverse Words in String

Problem: Reverse the order of words in a string.

Pseudocode:

```
FUNCTION reverseWords(string):
    words = SPLIT string by spaces
    reversedWords = []

    FOR i = words.length - 1 down to 0:
        IF words[i] is not empty:
            ADD words[i] to reversedWords

    RETURN JOIN reversedWords with single space
```

Explanation: Split string into words, then traverse word array in reverse order to build result. Handle multiple spaces by checking for empty strings. Time complexity: O(n), Space complexity: O(n).

Question 10: Check Balanced Parentheses

Problem: Check if parentheses in string are balanced.

Pseudocode:

```
FUNCTION isBalanced(string):
    count = 0

    FOR each character in string:
        IF character == '(':
            count = count + 1
        ELSE IF character == ')':
            count = count - 1
            IF count < 0:
                RETURN false

    RETURN count == 0
```

Explanation: Use counter for parentheses - increment for opening, decrement for closing. If counter goes negative or isn't zero at end, parentheses are unbalanced. Time complexity: O(n), Space complexity: O(1).

---

3. LINKED LISTS

Question 1: Reverse a Linked List

Problem: Reverse a singly linked list.

Pseudocode:

```
FUNCTION reverseLinkedList(head):
    previous = null
    current = head

    WHILE current != null:
        nextNode = current.next
        current.next = previous
        previous = current
        current = nextNode

    RETURN previous
```

Explanation: Use three pointers to iteratively reverse links. Previous starts as null, current at head. For each node, save next, reverse current's link to previous, then advance pointers. Time complexity: O(n), Space complexity: O(1).

Question 2: Find Middle of Linked List

Problem: Find the middle node of a linked list.

Pseudocode:

```
FUNCTION findMiddle(head):
    IF head == null:
        RETURN null

    slow = head
    fast = head

    WHILE fast != null AND fast.next != null:
        slow = slow.next
        fast = fast.next.next

    RETURN slow
```

Explanation: Floyd's tortoise and hare algorithm. Slow pointer moves one step, fast pointer moves two steps. When fast reaches end, slow is at middle. For even length, returns first of two middle nodes. Time complexity:

O(n), Space complexity: O(1).

Question 3: Detect Cycle in Linked List

Problem: Detect if there's a cycle in linked list.

Pseudocode:

```
FUNCTION hasCycle(head):
    IF head == null OR head.next == null:
        RETURN false

    slow = head
    fast = head

    WHILE fast != null AND fast.next != null:
        slow = slow.next
        fast = fast.next.next

        IF slow == fast:
            RETURN true

    RETURN false
```

Explanation: Floyd's cycle detection algorithm. If there's a cycle, fast and slow pointers will eventually meet. If no cycle, fast pointer reaches null. Time complexity: O(n), Space complexity: O(1).

Question 4: Remove Nth Node from End

Problem: Remove the nth node from the end of linked list.

Pseudocode:

```
FUNCTION removeNthFromEnd(head, n):
    dummy = CREATE new node with next pointing to head
    first = dummy
    second = dummy

    // Move first pointer n+1 steps ahead
    FOR i = 0 to n:
        first = first.next

    // Move both pointers until first reaches end
    WHILE first != null:
        first = first.next
        second = second.next

    // Remove the nth node
    second.next = second.next.next
```

```
    RETURN dummy.next
```

Explanation: Two pointer technique with dummy node. First pointer gets n+1 head start, then both move together. When first reaches end, second is at node before the one to delete. Time complexity: O(n), Space complexity: O(1).

Question 5: Merge Two Sorted Lists

Problem: Merge two sorted linked lists into one sorted list.

Pseudocode:

```
FUNCTION mergeSortedLists(list1, list2):
    dummy = CREATE new node
    current = dummy

    WHILE list1 != null AND list2 != null:
        IF list1.data <= list2.data:
            current.next = list1
            list1 = list1.next
        ELSE:
            current.next = list2
            list2 = list2.next
        current = current.next

    // Attach remaining nodes
    IF list1 != null:
        current.next = list1
    ELSE:
        current.next = list2

    RETURN dummy.next
```

Explanation: Use dummy node to simplify edge cases. Compare heads of both lists, attach smaller one to result, advance that pointer. Continue until one list is exhausted, then attach remaining nodes. Time complexity: O(n+m), Space complexity: O(1).

Question 6: Find Intersection of Two Lists

Problem: Find the node where two linked lists intersect.

Pseudocode:

```
FUNCTION findIntersection(headA, headB):
    IF headA == null OR headB == null:
        RETURN null

    pointerA = headA
```

```
    pointerB = headB

    WHILE pointerA != pointerB:
        pointerA = (pointerA == null) ? headB : pointerA.next
        pointerB = (pointerB == null) ? headA : pointerB.next

    RETURN pointerA
```

Explanation: Two pointers traverse both lists. When a pointer reaches end, it starts from other list's head. They will meet at intersection point or both become null if no intersection. This eliminates length difference. Time complexity: O(n+m), Space complexity: O(1).

Question 7: Remove Duplicates from Sorted List

Problem: Remove duplicates from a sorted linked list.

Pseudocode:

```
FUNCTION removeDuplicates(head):
    current = head

    WHILE current != null AND current.next != null:
        IF current.data == current.next.data:
            current.next = current.next.next
        ELSE:
            current = current.next

    RETURN head
```

Explanation: Since list is sorted, duplicates are adjacent. Compare current node with next node, if equal skip next node, otherwise advance current pointer. Time complexity: O(n), Space complexity: O(1).

Question 8: Check if Linked List is Palindrome

Problem: Check if values in linked list form a palindrome.

Pseudocode:

```
FUNCTION isPalindrome(head):
    IF head == null OR head.next == null:
        RETURN true

    // Find middle using slow-fast pointers
    slow = head
    fast = head
    WHILE fast.next != null AND fast.next.next != null:
        slow = slow.next
        fast = fast.next.next
```

```
    // Reverse second half
    secondHalf = reverseList(slow.next)

    // Compare first half with reversed second half
    firstHalf = head
    WHILE secondHalf != null:
        IF firstHalf.data != secondHalf.data:
            RETURN false
        firstHalf = firstHalf.next
        secondHalf = secondHalf.next

    RETURN true
```

Explanation: Find middle, reverse second half, compare both halves. This modifies the list but uses O(1) space. Alternative: use stack to store first half. Time complexity: O(n), Space complexity: O(1).

Question 9: Add Two Numbers Represented as Lists

Problem: Add two numbers where digits are stored in reverse order in linked lists.

Pseudocode:

```
FUNCTION addTwoNumbers(list1, list2):
    dummy = CREATE new node
    current = dummy
    carry = 0

    WHILE list1 != null OR list2 != null OR carry != 0:
        sum = carry

        IF list1 != null:
            sum = sum + list1.data
            list1 = list1.next

        IF list2 != null:
            sum = sum + list2.data
            list2 = list2.next

        carry = sum / 10
        current.next = CREATE new node with data (sum % 10)
        current = current.next

    RETURN dummy.next
```

Explanation: Simulate addition digit by digit, handling carry. Continue until both lists are exhausted and no carry remains. Each digit is sum of corresponding digits plus carry from previous position. Time complexity: O(max(n,m)), Space complexity: O(max(n,m)).

Question 10: Rotate Linked List

Problem: Rotate linked list to the right by k places.

Pseudocode:

```
FUNCTION rotateRight(head, k):
    IF head == null OR head.next == null OR k == 0:
        RETURN head

    // Find length and make it circular
    length = 1
    tail = head
    WHILE tail.next != null:
        tail = tail.next
        length = length + 1

    tail.next = head  // Make circular

    // Find new tail (length - k % length - 1 steps from head)
    k = k % length
    stepsToNewTail = length - k
    newTail = head

    FOR i = 1 to stepsToNewTail - 1:
        newTail = newTail.next

    newHead = newTail.next
    newTail.next = null

    RETURN newHead
```

Explanation: Make list circular, find new head and tail positions based on rotation amount, then break the circle. Handle k > length by using modulo. Time complexity: O(n), Space complexity: O(1).

---

4. STACKS

Question 1: Implement Stack using Array

Problem: Implement basic stack operations using array.

Pseudocode:

```
CLASS Stack:
    INITIALIZE:
        array = ARRAY of fixed size
        top = -1
        maxSize = array.length

    FUNCTION push(element):
        IF top == maxSize - 1:
```

```
            PRINT "Stack Overflow"
            RETURN
        top = top + 1
        array[top] = element

    FUNCTION pop():
        IF top == -1:
            PRINT "Stack Underflow"
            RETURN null
        element = array[top]
        top = top - 1
        RETURN element

    FUNCTION peek():
        IF top == -1:
            RETURN null
        RETURN array[top]

    FUNCTION isEmpty():
        RETURN top == -1
```

Explanation: Array-based stack with top pointer. Push increments top and adds element, pop returns top element and decrements top. Check bounds to avoid overflow/underflow. Time complexity: O(1) for all operations, Space complexity: O(n).

Question 2: Valid Parentheses

Problem: Check if string has valid parentheses (including [], {}, ()).

Pseudocode:

```
FUNCTION isValidParentheses(string):
    stack = CREATE empty stack

    FOR each character in string:
        IF character is opening bracket ['(', '[', '{']:
            PUSH character to stack
        ELSE IF character is closing bracket [')', ']', '}']:
            IF stack is empty:
                RETURN false

            top = POP from stack
            IF NOT isMatchingPair(top, character):
                RETURN false

    RETURN stack is empty

FUNCTION isMatchingPair(opening, closing):
    RETURN (opening == '(' AND closing == ')') OR
           (opening == '[' AND closing == ']') OR
           (opening == '{' AND closing == '}')
```

Explanation: Push opening brackets onto stack, for closing brackets check if they match most recent opening bracket. Stack should be empty at end for valid parentheses. Time complexity: O(n), Space complexity: O(n).

Question 3: Next Greater Element

Problem: For each element in array, find the next greater element to its right.

Pseudocode:

```
FUNCTION nextGreaterElement(array):
    n = array.length
    result = ARRAY of size n filled with -1
    stack = CREATE empty stack

    FOR i = 0 to n - 1:
        WHILE stack is not empty AND array[i] > array[stack.top()]:
            index = POP from stack
            result[index] = array[i]

        PUSH i to stack

    RETURN result
```

Explanation: Use stack to store indices of elements for which we haven't found next greater element. When we find a larger element, it's the answer for all smaller elements in stack. Time complexity: O(n), Space complexity: O(n).

Question 4: Evaluate Postfix Expression

Problem: Evaluate arithmetic expression in postfix notation.

Pseudocode:

```
FUNCTION evaluatePostfix(expression):
    stack = CREATE empty stack

    FOR each token in expression:
        IF token is number:
            PUSH token to stack
        ELSE IF token is operator ['+', '-', '*', '/']:
            IF stack has less than 2 elements:
                RETURN error

            operand2 = POP from stack
            operand1 = POP from stack
            result = PERFORM operation(operand1, token, operand2)
            PUSH result to stack

    IF stack has exactly 1 element:
```

```
        RETURN POP from stack
    ELSE:
        RETURN error
```

Explanation: Push operands onto stack, when operator is encountered pop two operands, perform operation, push result back. Final stack should have one element (the result). Time complexity: O(n), Space complexity: O(n).

Question 5: Minimum Stack

Problem: Design stack that supports push, pop, top, and retrieving minimum element in constant time.

Pseudocode:

```
CLASS MinStack:
    INITIALIZE:
        mainStack = CREATE empty stack
        minStack = CREATE empty stack

    FUNCTION push(element):
        PUSH element to mainStack

        IF minStack is empty OR element <= minStack.top():
            PUSH element to minStack

    FUNCTION pop():
        IF mainStack is empty:
            RETURN error

        element = POP from mainStack
        IF element == minStack.top():
            POP from minStack

        RETURN element

    FUNCTION top():
        RETURN mainStack.top()

    FUNCTION getMin():
        RETURN minStack.top()
```

Explanation: Use auxiliary stack to track minimum elements. Push to min stack only when new element is less than or equal to current minimum. Pop from min stack only when popped element equals current minimum. Time complexity: O(1) for all operations, Space complexity: O(n).

Question 6: Largest Rectangle in Histogram

Problem: Find area of largest rectangle that can be formed in histogram.

Pseudocode:

```
FUNCTION largestRectangle(heights):
    stack = CREATE empty stack
    maxArea = 0

    FOR i = 0 to heights.length - 1:
        WHILE stack is not empty AND heights[i] < heights[stack.top()]:
            height = heights[POP from stack]
            width = (stack is empty) ? i : (i - stack.top() - 1)
            area = height * width
            maxArea = MAX(maxArea, area)

        PUSH i to stack

    WHILE stack is not empty:
        height = heights[POP from stack]
        width = (stack is empty) ? heights.length : (heights.length - stack.top()
- 1)
        area = height * width
        maxArea = MAX(maxArea, area)

    RETURN maxArea
```

Explanation: Use stack to store indices of bars in increasing order of heights. When we find smaller bar, calculate area with each popped bar as smallest bar. Width is determined by current position and previous bar in stack. Time complexity: O(n), Space complexity: O(n).

Question 7: Sort Stack using Another Stack

Problem: Sort a stack using only one additional stack.

Pseudocode:

```
FUNCTION sortStack(inputStack):
    tempStack = CREATE empty stack

    WHILE inputStack is not empty:
        temp = POP from inputStack

        WHILE tempStack is not empty AND tempStack.top() > temp:
            PUSH POP(tempStack) to inputStack

        PUSH temp to tempStack

    // Move all elements back to input stack
    WHILE tempStack is not empty:
        PUSH POP(tempStack) to inputStack

    RETURN inputStack
```

Explanation: Use temporary stack to sort elements. For each element from input stack, find its correct position in temp stack by moving larger elements back to input stack. Time complexity: $O(n^2)$, Space complexity: O(n).

Question 8: Balanced Brackets with Stack

Problem: Check if brackets are balanced considering multiple types and nested structures.

Pseudocode:

```
FUNCTION isBalanced(expression):
    stack = CREATE empty stack
    brackets = {
        ')': '(',
        ']': '[',
        '}': '{'
    }

    FOR each character in expression:
        IF character in ['(', '[', '{']:
            PUSH character to stack
        ELSE IF character in [')', ']', '}']:
            IF stack is empty:
                RETURN false

            IF POP from stack != brackets[character]:
                RETURN false

    RETURN stack is empty
```

Explanation: Enhanced version of parentheses checking supporting multiple bracket types. Use mapping from closing to opening brackets for easy validation. Time complexity: O(n), Space complexity: O(n).

Question 9: Reverse Stack using Recursion

Problem: Reverse stack using recursion without using extra space.

Pseudocode:

```
FUNCTION reverseStack(stack):
    IF stack is not empty:
        temp = POP from stack
        reverseStack(stack)
        insertAtBottom(stack, temp)

FUNCTION insertAtBottom(stack, element):
    IF stack is empty:
        PUSH element to stack
    ELSE:
        temp = POP from stack
```

```
        insertAtBottom(stack, element)
        PUSH temp to stack
```

Explanation: Recursive solution that removes all elements, reverses remaining stack, then inserts element at bottom. Uses call stack instead of auxiliary data structure. Time complexity: $O(n^2)$, Space complexity: O(n) for recursion.

Question 10: Stock Span Problem

Problem: Calculate span of stock prices (number of consecutive days before current day with price less than or equal to current day).

Pseudocode:

```
FUNCTION calculateSpan(prices):
    n = prices.length
    span = ARRAY of size n
    stack = CREATE empty stack

    FOR i = 0 to n - 1:
        WHILE stack is not empty AND prices[stack.top()] <= prices[i]:
            POP from stack

        span[i] = (stack is empty) ? (i + 1) : (i - stack.top())
        PUSH i to stack

    RETURN span
```

Explanation: Use stack to store indices of days in decreasing order of prices. For each day, pop all days with lower or equal prices, then span is distance to previous higher price day (or beginning if stack empty). Time complexity: O(n), Space complexity: O(n).

---

5. QUEUES

Question 1: Implement Queue using Array

Problem: Implement basic queue operations using circular array.

Pseudocode:

```
CLASS Queue:
    INITIALIZE:
        array = ARRAY of fixed size
        front = 0
        rear = -1
        size = 0
        maxSize = array.length
```

```
    FUNCTION enqueue(element):
        IF size == maxSize:
            PRINT "Queue Overflow"
            RETURN
        rear = (rear + 1) % maxSize
        array[rear] = element
        size = size + 1

    FUNCTION dequeue():
        IF size == 0:
            PRINT "Queue Underflow"
            RETURN null
        element = array[front]
        front = (front + 1) % maxSize
        size = size - 1
        RETURN element

    FUNCTION peek():
        IF size == 0:
            RETURN null
        RETURN array[front]

    FUNCTION isEmpty():
        RETURN size == 0
```

Explanation: Circular array implementation to efficiently use space. Use modulo arithmetic for wrap-around. Track front, rear, and size to handle all edge cases. Time complexity: O(1) for all operations, Space complexity: O(n).

Question 2: Implement Queue using Stacks

Problem: Implement queue using two stacks.

Pseudocode:

```
CLASS QueueUsingStacks:
    INITIALIZE:
        stack1 = CREATE empty stack  // For enqueue
        stack2 = CREATE empty stack  // For dequeue

    FUNCTION enqueue(element):
        PUSH element to stack1

    FUNCTION dequeue():
        IF both stacks are empty:
            PRINT "Queue is empty"
            RETURN null

        IF stack2 is empty:
            WHILE stack1 is not empty:
                PUSH POP(stack1) to stack2
```

```
        RETURN POP from stack2

    FUNCTION peek():
        IF both stacks are empty:
            RETURN null

        IF stack2 is empty:
            WHILE stack1 is not empty:
                PUSH POP(stack1) to stack2

        RETURN stack2.top()
```

Explanation: Use two stacks - one for enqueue (stack1), one for dequeue (stack2). Transfer elements from stack1 to stack2 only when stack2 is empty and dequeue is needed. Amortized time complexity: O(1), Space complexity: O(n).

Question 3: Generate Binary Numbers using Queue

Problem: Generate binary representations of numbers from 1 to n.

Pseudocode:

```
FUNCTION generateBinaryNumbers(n):
    result = []
    queue = CREATE empty queue

    ENQUEUE "1" to queue

    FOR i = 1 to n:
        binary = DEQUEUE from queue
        ADD binary to result

        ENQUEUE binary + "0" to queue
        ENQUEUE binary + "1" to queue

    RETURN result
```

Explanation: Start with "1" in queue. For each number, dequeue current binary string, add to result, then enqueue its children by appending "0" and "1". This generates binary numbers in sequence. Time complexity: O(n), Space complexity: O(n).

Question 4: First Non-Repeating Character in Stream

Problem: Find first non-repeating character in a stream of characters.

Pseudocode:

```

```

```
CLASS FirstNonRepeating:
    INITIALIZE:
        queue = CREATE empty queue
        charCount = ARRAY of size 256 initialized to 0

    FUNCTION processCharacter(character):
        charCount[ASCII(character)] = charCount[ASCII(character)] + 1
        ENQUEUE character to queue

        WHILE queue is not empty AND charCount[ASCII(queue.front())] > 1:
            DEQUEUE from queue

        IF queue is empty:
            RETURN null
        ELSE:
            RETURN queue.front()
```

Explanation: Use queue to maintain order of characters and frequency array to track counts. Remove characters from front of queue if their count becomes greater than 1. Time complexity: O(1) per character, Space complexity: O(n).

Question 5: Sliding Window Maximum

Problem: Find maximum element in every sliding window of size k.

Pseudocode:

```
FUNCTION slidingWindowMaximum(array, k):
    result = []
    deque = CREATE empty deque  // Stores indices

    FOR i = 0 to array.length - 1:
        // Remove indices outside current window
        WHILE deque is not empty AND deque.front() <= i - k:
            REMOVE front from deque

        // Remove indices of elements smaller than current
        WHILE deque is not empty AND array[deque.back()] <= array[i]:
            REMOVE back from deque

        ADD i to back of deque

        // Add maximum to result if window is complete
        IF i >= k - 1:
            ADD array[deque.front()] to result

    RETURN result
```

Explanation: Use deque to store indices in decreasing order of their values. Front always contains index of maximum element in current window. Remove outdated indices and smaller elements. Time complexity: O(n), Space complexity: O(k).

Question 6: Reverse First K Elements of Queue

Problem: Reverse the first k elements of a queue.

Pseudocode:

```
FUNCTION reverseFirstK(queue, k):
    IF k <= 0 OR k > queue.size():
        RETURN queue

    stack = CREATE empty stack

    // Dequeue first k elements and push to stack
    FOR i = 1 to k:
        PUSH DEQUEUE(queue) to stack

    // Pop from stack and enqueue back to queue
    WHILE stack is not empty:
        ENQUEUE POP(stack) to queue

    // Move remaining (n-k) elements to back
    FOR i = 1 to (queue.size() - k):
        ENQUEUE DEQUEUE(queue) to queue

    RETURN queue
```

Explanation: Use stack to reverse first k elements, then rotate remaining elements to maintain their relative order. Time complexity: O(n), Space complexity: O(k).

Question 7: Interleave First and Second Half of Queue

Problem: Interleave elements from first and second half of queue.

Pseudocode:

```
FUNCTION interleaveQueue(queue):
    n = queue.size()
    IF n % 2 != 0:
        RETURN error  // Queue size must be even

    stack = CREATE empty stack
    halfSize = n / 2

    // Push first half to stack
    FOR i = 1 to halfSize:
        PUSH DEQUEUE(queue) to stack
```

```
    // Pop from stack and enqueue (reverses first half)
    WHILE stack is not empty:
        ENQUEUE POP(stack) to queue

    // Move first half to back
    FOR i = 1 to halfSize:
        ENQUEUE DEQUEUE(queue) to queue

    // Push first half to stack again
    FOR i = 1 to halfSize:
        PUSH DEQUEUE(queue) to stack

    // Interleave: pop from stack and dequeue, then enqueue both
    WHILE stack is not empty:
        ENQUEUE POP(stack) to queue
        ENQUEUE DEQUEUE(queue) to queue

    RETURN queue
```

Explanation: Complex manipulation using stack and queue operations to achieve interleaving. Multiple steps to reverse and rearrange elements properly. Time complexity: O(n), Space complexity: O(n/2).

Question 8: Queue using Linked List

Problem: Implement queue using linked list.

Pseudocode:

```
CLASS Node:
    data = null
    next = null

CLASS QueueLinkedList:
    INITIALIZE:
        front = null
        rear = null

    FUNCTION enqueue(element):
        newNode = CREATE new Node with data = element

        IF rear == null:
            front = rear = newNode
        ELSE:
            rear.next = newNode
            rear = newNode

    FUNCTION dequeue():
        IF front == null:
            PRINT "Queue is empty"
            RETURN null
```

```
        element = front.data
        front = front.next

        IF front == null:
            rear = null

        RETURN element

    FUNCTION peek():
        IF front == null:
            RETURN null
        RETURN front.data

    FUNCTION isEmpty():
        RETURN front == null
```

Explanation: Linked list implementation allows dynamic size. Enqueue at rear, dequeue from front. Handle edge case when queue becomes empty (both front and rear become null). Time complexity: O(1) for all operations, Space complexity: O(n).

Question 9: Circular Queue Implementation

Problem: Implement circular queue with fixed size.

Pseudocode:

```
CLASS CircularQueue:
    INITIALIZE:
        array = ARRAY of size k
        front = 0
        rear = -1
        size = 0
        capacity = k

    FUNCTION enqueue(element):
        IF isFull():
            RETURN false
        rear = (rear + 1) % capacity
        array[rear] = element
        size = size + 1
        RETURN true

    FUNCTION dequeue():
        IF isEmpty():
            RETURN false
        front = (front + 1) % capacity
        size = size - 1
        RETURN true

    FUNCTION getFront():
```

```
        IF isEmpty():
            RETURN -1
        RETURN array[front]

    FUNCTION getRear():
        IF isEmpty():
            RETURN -1
        RETURN array[rear]

    FUNCTION isEmpty():
        RETURN size == 0

    FUNCTION isFull():
        RETURN size == capacity
```

Explanation: Fixed-size circular queue using array with wrap-around using modulo. Track size separately to distinguish between empty and full states. Time complexity: O(1) for all operations, Space complexity: O(k).

Question 10: Level Order Traversal using Queue

Problem: Print binary tree nodes level by level using queue.

Pseudocode:

```
FUNCTION levelOrderTraversal(root):
    IF root == null:
        RETURN

    queue = CREATE empty queue
    ENQUEUE root to queue

    WHILE queue is not empty:
        levelSize = queue.size()

        FOR i = 1 to levelSize:
            node = DEQUEUE from queue
            PRINT node.data

            IF node.left != null:
                ENQUEUE node.left to queue

            IF node.right != null:
                ENQUEUE node.right to queue

        PRINT newline  // Separate levels
```

Explanation: BFS traversal using queue. Process all nodes at current level before moving to next level by tracking level size. Enqueue children of current level nodes for next iteration. Time complexity: O(n), Space complexity: O(w) where w is maximum width.

6. TREES

Question 1: Binary Tree Traversals

Problem: Implement inorder, preorder, and postorder traversals.

Pseudocode:

```
// Inorder: Left -> Root -> Right
FUNCTION inorderTraversal(root):
    IF root != null:
        inorderTraversal(root.left)
        PRINT root.data
        inorderTraversal(root.right)

// Preorder: Root -> Left -> Right
FUNCTION preorderTraversal(root):
    IF root != null:
        PRINT root.data
        preorderTraversal(root.left)
        preorderTraversal(root.right)

// Postorder: Left -> Right -> Root
FUNCTION postorderTraversal(root):
    IF root != null:
        postorderTraversal(root.left)
        postorderTraversal(root.right)
        PRINT root.data
```

Explanation: Three fundamental tree traversal methods. Inorder gives sorted sequence for BST, preorder useful for copying tree, postorder for deletion/cleanup. Time complexity: O(n), Space complexity: O(h) where h is height.

Question 2: Height of Binary Tree

Problem: Find the height/depth of a binary tree.

Pseudocode:

```
FUNCTION treeHeight(root):
    IF root == null:
        RETURN 0

    leftHeight = treeHeight(root.left)
    rightHeight = treeHeight(root.right)

    RETURN 1 + MAX(leftHeight, rightHeight)
```

Explanation: Recursive approach calculating height as 1 plus maximum of left and right subtree heights. Base case returns 0 for null node. Time complexity: O(n), Space complexity: O(h) for recursion stack.

Question 3: Check if Binary Tree is Balanced

Problem: Check if binary tree is height-balanced (height difference between subtrees ≤ 1).

Pseudocode:

```
FUNCTION isBalanced(root):
    RETURN checkBalance(root) != -1

FUNCTION checkBalance(root):
    IF root == null:
        RETURN 0

    leftHeight = checkBalance(root.left)
    IF leftHeight == -1:
        RETURN -1

    rightHeight = checkBalance(root.right)
    IF rightHeight == -1:
        RETURN -1

    IF ABS(leftHeight - rightHeight) > 1:
        RETURN -1

    RETURN 1 + MAX(leftHeight, rightHeight)
```

Explanation: Optimized approach that checks balance and calculates height simultaneously. Returns -1 if subtree is unbalanced, otherwise returns height. Early termination on imbalance. Time complexity: O(n), Space complexity: O(h).

Question 4: Diameter of Binary Tree

Problem: Find the longest path between any two nodes in binary tree.

Pseudocode:

```
FUNCTION treeDiameter(root):
    maxDiameter = 0

    FUNCTION calculateHeight(node):
        IF node == null:
            RETURN 0

        leftHeight = calculateHeight(node.left)
        rightHeight = calculateHeight(node.right)

        currentDiameter = leftHeight + rightHeight
```

```
        maxDiameter = MAX(maxDiameter, currentDiameter)

        RETURN 1 + MAX(leftHeight, rightHeight)

    calculateHeight(root)
    RETURN maxDiameter
```

Explanation: For each node, diameter passing through it is sum of left and right subtree heights. Global maximum tracks the answer. Uses nested function to maintain state. Time complexity: O(n), Space complexity: O(h).

Question 5: Lowest Common Ancestor in Binary Tree

Problem: Find the lowest common ancestor of two nodes in binary tree.

Pseudocode:

```
FUNCTION findLCA(root, node1, node2):
    IF root == null:
        RETURN null

    IF root == node1 OR root == node2:
        RETURN root

    leftLCA = findLCA(root.left, node1, node2)
    rightLCA = findLCA(root.right, node1, node2)

    IF leftLCA != null AND rightLCA != null:
        RETURN root

    RETURN (leftLCA != null) ? leftLCA : rightLCA
```

Explanation: If current node is one of target nodes, return it. If both subtrees return non-null, current node is LCA. Otherwise, return whichever subtree found a target node. Time complexity: O(n), Space complexity: O(h).

Question 6: Convert Binary Tree to Mirror

Problem: Convert binary tree to its mirror image.

Pseudocode:

```
FUNCTION mirrorTree(root):
    IF root == null:
        RETURN null

    // Recursively mirror subtrees
    mirrorTree(root.left)
    mirrorTree(root.right)
```

```
    // Swap left and right children
    temp = root.left
    root.left = root.right
    root.right = temp

    RETURN root
```

Explanation: Post-order approach - first mirror subtrees, then swap children of current node. Could also use pre-order (swap first, then recurse). Time complexity: O(n), Space complexity: O(h).

Question 7: Binary Search Tree Operations

Problem: Implement search, insert, and delete operations in BST.

Pseudocode:

```
FUNCTION searchBST(root, key):
    IF root == null OR root.data == key:
        RETURN root

    IF key < root.data:
        RETURN searchBST(root.left, key)
    ELSE:
        RETURN searchBST(root.right, key)

FUNCTION insertBST(root, key):
    IF root == null:
        RETURN CREATE new node with data = key

    IF key < root.data:
        root.left = insertBST(root.left, key)
    ELSE IF key > root.data:
        root.right = insertBST(root.right, key)

    RETURN root

FUNCTION deleteBST(root, key):
    IF root == null:
        RETURN root

    IF key < root.data:
        root.left = deleteBST(root.left, key)
    ELSE IF key > root.data:
        root.right = deleteBST(root.right, key)
    ELSE:
        // Node to delete found
        IF root.left == null:
            RETURN root.right
        ELSE IF root.right == null:
            RETURN root.left
```

```
        // Node has two children
        successor = findMin(root.right)
        root.data = successor.data
        root.right = deleteBST(root.right, successor.data)

    RETURN root

FUNCTION findMin(root):
    WHILE root.left != null:
        root = root.left
    RETURN root
```

Explanation: BST property: left subtree < root < right subtree. Search/insert follow this property. Delete handles three cases: no children, one child, two children (replace with inorder successor). Time complexity: O(h), Space complexity: O(h).

Question 8: Validate Binary Search Tree

Problem: Check if binary tree is valid BST.

Pseudocode:

```
FUNCTION isValidBST(root):
    RETURN validateBST(root, -infinity, +infinity)

FUNCTION validateBST(node, minVal, maxVal):
    IF node == null:
        RETURN true

    IF node.data <= minVal OR node.data >= maxVal:
        RETURN false

    RETURN validateBST(node.left, minVal, node.data) AND
           validateBST(node.right, node.data, maxVal)
```

Explanation: Use bounds checking approach. Each node must be within valid range determined by ancestors. Left subtree nodes must be less than current node, right subtree nodes must be greater. Time complexity: O(n), Space complexity: O(h).

Question 9: Path Sum in Binary Tree

Problem: Check if there exists a path from root to leaf with given sum.

Pseudocode:

```
FUNCTION hasPathSum(root, targetSum):
    IF root == null:
        RETURN false
```

```
    // Leaf node case
    IF root.left == null AND root.right == null:
        RETURN root.data == targetSum

    remainingSum = targetSum - root.data

    RETURN hasPathSum(root.left, remainingSum) OR
           hasPathSum(root.right, remainingSum)
```

Explanation: Recursive approach reducing target sum by current node value. Base case checks if leaf node value equals remaining sum. Time complexity: O(n), Space complexity: O(h).

Question 10: Serialize and Deserialize Binary Tree

Problem: Convert binary tree to string representation and reconstruct from string.

Pseudocode:

```
FUNCTION serialize(root):
    result = []

    FUNCTION serializeHelper(node):
        IF node == null:
            ADD "null" to result
            RETURN

        ADD node.data to result
        serializeHelper(node.left)
        serializeHelper(node.right)

    serializeHelper(root)
    RETURN JOIN result with ","

FUNCTION deserialize(data):
    values = SPLIT data by ","
    index = 0

    FUNCTION deserializeHelper():
        IF values[index] == "null":
            index = index + 1
            RETURN null

        node = CREATE new node with data = values[index]
        index = index + 1

        node.left = deserializeHelper()
        node.right = deserializeHelper()

        RETURN node

    RETURN deserializeHelper()
```

Explanation: Preorder traversal for serialization with null markers. Deserialization reconstructs tree using same order. Index tracks current position in value array. Time complexity: O(n), Space complexity: O(n).

---

7. BINARY SEARCH

Question 1: Basic Binary Search

Problem: Find target element in sorted array.

Pseudocode:

```
FUNCTION binarySearch(array, target):
    left = 0
    right = array.length - 1

    WHILE left <= right:
        mid = left + (right - left) / 2

        IF array[mid] == target:
            RETURN mid
        ELSE IF array[mid] < target:
            left = mid + 1
        ELSE:
            right = mid - 1

    RETURN -1
```

Explanation: Divide and conquer approach. Compare middle element with target, eliminate half of search space based on comparison. Use `left + (right - left) / 2` to avoid overflow. Time complexity: O(log n), Space complexity: O(1).

Question 2: Find First and Last Position

Problem: Find first and last occurrence of target in sorted array with duplicates.

Pseudocode:

```
FUNCTION findFirstAndLast(array, target):
    first = findFirst(array, target)
    last = findLast(array, target)
    RETURN [first, last]

FUNCTION findFirst(array, target):
    left = 0
    right = array.length - 1
    result = -1

    WHILE left <= right:
```

```
        mid = left + (right - left) / 2

        IF array[mid] == target:
            result = mid
            right = mid - 1  // Continue searching in left half
        ELSE IF array[mid] < target:
            left = mid + 1
        ELSE:
            right = mid - 1

    RETURN result

FUNCTION findLast(array, target):
    left = 0
    right = array.length - 1
    result = -1

    WHILE left <= right:
        mid = left + (right - left) / 2

        IF array[mid] == target:
            result = mid
            left = mid + 1  // Continue searching in right half
        ELSE IF array[mid] < target:
            left = mid + 1
        ELSE:
            right = mid - 1

    RETURN result
```

Explanation: Modified binary search that continues searching even after finding target. For first occurrence, search left half after finding match. For last occurrence, search right half. Time complexity: O(log n), Space complexity: O(1).

Question 3: Search in Rotated Sorted Array

Problem: Search target in rotated sorted array.

Pseudocode:

```
FUNCTION searchRotated(array, target):
    left = 0
    right = array.length - 1

    WHILE left <= right:
        mid = left + (right - left) / 2

        IF array[mid] == target:
            RETURN mid

        // Check which half is sorted
```

```
        IF array[left] <= array[mid]:  // Left half is sorted
            IF target >= array[left] AND target < array[mid]:
                right = mid - 1
            ELSE:
                left = mid + 1
        ELSE:  // Right half is sorted
            IF target > array[mid] AND target <= array[right]:
                left = mid + 1
            ELSE:
                right = mid - 1

    RETURN -1
```

Explanation: One half is always sorted in rotated array. Identify sorted half, check if target lies in it, then search accordingly. Handle edge cases with duplicate elements carefully. Time complexity: O(log n), Space complexity: O(1).

Question 4: Find Peak Element

Problem: Find any peak element in array (element greater than its neighbors).

Pseudocode:

```
FUNCTION findPeakElement(array):
    left = 0
    right = array.length - 1

    WHILE left < right:
        mid = left + (right - left) / 2

        IF array[mid] > array[mid + 1]:
            right = mid  // Peak is in left half (including mid)
        ELSE:
            left = mid + 1  // Peak is in right half

    RETURN left
```

Explanation: If middle element is greater than next element, peak exists in left half (including middle). Otherwise, peak exists in right half. Always converges to a peak. Time complexity: O(log n), Space complexity: O(1).

Question 5: Square Root using Binary Search

Problem: Find integer square root of a number.

Pseudocode:

```
FUNCTION sqrt(x):
    IF x < 2:
```

```
        RETURN x

    left = 2
    right = x / 2

    WHILE left <= right:
        mid = left + (right - left) / 2
        square = mid * mid

        IF square == x:
            RETURN mid
        ELSE IF square < x:
            left = mid + 1
        ELSE:
            right = mid - 1

    RETURN right  // Return floor value
```

Explanation: Binary search on answer space [2, x/2]. For x ≥ 2, square root is at most x/2. Compare square of middle with x to determine search direction. Time complexity: O(log x), Space complexity: O(1).

Question 6: Search Insert Position

Problem: Find index where target should be inserted in sorted array.

Pseudocode:

```
FUNCTION searchInsert(array, target):
    left = 0
    right = array.length - 1

    WHILE left <= right:
        mid = left + (right - left) / 2

        IF array[mid] == target:
            RETURN mid
        ELSE IF array[mid] < target:
            left = mid + 1
        ELSE:
            right = mid - 1

    RETURN left
```

Explanation: Standard binary search with modification - when target not found, left pointer gives insertion position. This maintains sorted order after insertion. Time complexity: O(log n), Space complexity: O(1).

Question 7: Find Minimum in Rotated Sorted Array

Problem: Find minimum element in rotated sorted array.

Pseudocode:

```
FUNCTION findMin(array):
    left = 0
    right = array.length - 1

    WHILE left < right:
        mid = left + (right - left) / 2

        IF array[mid] > array[right]:
            left = mid + 1  // Minimum is in right half
        ELSE:
            right = mid  // Minimum is in left half (including mid)

    RETURN array[left]
```

Explanation: Compare middle with rightmost element. If middle > right, minimum is in right half. Otherwise, minimum is in left half including middle. Converges to minimum element. Time complexity: O(log n), Space complexity: O(1).

Question 8: Count Occurrences in Sorted Array

Problem: Count number of occurrences of target in sorted array.

Pseudocode:

```
FUNCTION countOccurrences(array, target):
    first = findFirst(array, target)
    IF first == -1:
        RETURN 0

    last = findLast(array, target)
    RETURN last - first + 1

// Use findFirst and findLast functions from Question 2
```

Explanation: Find first and last positions of target using binary search, then calculate count as difference plus one. If target not found, return 0. Time complexity: O(log n), Space complexity: O(1).

Question 9: Search in 2D Matrix

Problem: Search target in matrix where each row and column is sorted.

Pseudocode:

```
FUNCTION searchMatrix(matrix, target):
    IF matrix is empty:
        RETURN false
```

```
    row = 0
    col = matrix[0].length - 1

    WHILE row < matrix.length AND col >= 0:
        IF matrix[row][col] == target:
            RETURN true
        ELSE IF matrix[row][col] > target:
            col = col - 1  // Move left
        ELSE:
            row = row + 1  // Move down

    RETURN false
```

Explanation: Start from top-right corner. If current element is greater than target, move left. If smaller, move down. This eliminates one row or column in each step. Time complexity: O(m + n), Space complexity: O(1).

Question 10: Capacity to Ship Packages

Problem: Find minimum capacity needed to ship all packages within given days.

Pseudocode:

```
FUNCTION shipWithinDays(weights, days):
    left = MAX(weights)  // Minimum capacity needed
    right = SUM(weights)  // Maximum capacity needed

    WHILE left < right:
        mid = left + (right - left) / 2

        IF canShip(weights, days, mid):
            right = mid
        ELSE:
            left = mid + 1

    RETURN left

FUNCTION canShip(weights, days, capacity):
    currentWeight = 0
    daysNeeded = 1

    FOR each weight in weights:
        IF currentWeight + weight > capacity:
            daysNeeded = daysNeeded + 1
            currentWeight = weight
        ELSE:
            currentWeight = currentWeight + weight

    RETURN daysNeeded <= days
```

Explanation: Binary search on capacity range. Minimum capacity is maximum single package weight, maximum is sum of all weights. Check if given capacity allows shipping within required days. Time complexity: O(n * log(sum)), Space complexity: O(1).