Programming Interview Guide: Arrays

1 Arrays

Arrays are fundamental data structures that store elements in contiguous memory locations. Common problems involve manipulation, searching, or sorting. This guide provides two example problems with detailed explanations and Python code implementations.

1.1 Example 1: Two Sum

Problem Statement: Given an array of integers `nums` and an integer `target`, find two numbers such that they add up to `target`. Return their indices. Assume exactly one solution exists.

Solution Approach:

- Use a hash map to store numbers and their indices.
- Iterate through the array, checking if `target - nums[i]` exists in the hash map.
- If found, return the current index and the stored index.

Time Complexity: $O(n)$, where n is the array length.
Space Complexity: $O(n)$ for the hash map.

```python
def twoSum(nums, target):
    seen = {}
    for i, num in enumerate(nums):
        complement = target - num
        if complement in seen:
            return [seen[complement], i]
        seen[num] = i
    return []
```

Explanation: The function uses a dictionary to store each number and its index. For each number, it calculates the complement (`target - num`) and checks if it exists in the dictionary. If it does, it returns the indices. Otherwise, it adds the current number and index to the dictionary.

1.2 Example 2: Maximum Subarray

Problem Statement: Given an integer array `nums`, find the contiguous subarray with the largest sum and return its sum.

Solution Approach:

- Use Kadane's algorithm to track the maximum sum ending at each index.
- Maintain a global maximum sum.
- Update the current sum by taking the maximum of the current element and the sum including it.

Time Complexity: $O(n)$, where n is the array length.
Space Complexity: $O(1)$.

```
def maxSubArray(nums):
    max_sum = nums[0]
    current_sum = nums[0]
    for num in nums[1:]:
        current_sum = max(num, current_sum + num)
        max_sum = max(max_sum, current_sum)
    return max_sum
```

Explanation: Kadane's algorithm iterates through the array, deciding at each step whether to start a new subarray at the current element or extend the existing subarray. The maximum sum is updated whenever a larger sum is found.

Programming Interview Guide: Strings

1 Strings

String problems often involve manipulation, pattern matching, or substring searches. This guide provides two example problems with detailed explanations and Python code implementations.

1.1 Example 1: Longest Substring Without Repeating Characters

Problem Statement: Given a string `s`, find the length of the longest substring without repeating characters.

Solution Approach:

- Use a sliding window with a hash map to track character positions.
- Move the right pointer, updating the window start if a repeat is found.
- Track the maximum length of the window.

Time Complexity: $O(n)$, where n is the string length.
Space Complexity: $O(\min(m, n))$, where m is the size of the character set.

```python
def lengthOfLongestSubstring(s):
    seen = {}
    max_length = 0
    start = 0
    for end in range(len(s)):
        if s[end] in seen and seen[s[end]] >= start:
            start = seen[s[end]] + 1
        else:
            max_length = max(max_length, end - start + 1)
        seen[s[end]] = end
    return max_length
```

Explanation: The sliding window tracks the longest substring without repeats. When a repeating character is found, the window's start is moved past the previous occurrence. The hash map stores the latest index of each character.

1.2 Example 2: Valid Palindrome

Problem Statement: Given a string `s`, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.

Solution Approach:

- Use two pointers (left and right) to compare characters.
- Skip non-alphanumeric characters and convert to lowercase.
- Check if characters match until pointers meet.

Time Complexity: $O(n)$, where n is the string length.
Space Complexity: $O(1)$.

```python
def isPalindrome(s):
    left, right = 0, len(s) - 1
    while left < right:
        while left < right and not s[left].isalnum():
            left += 1
        while left < right and not s[right].isalnum():
            right -= 1
        if s[left].lower() != s[right].lower():
            return False
        left += 1
        right -= 1
    return True
```

Explanation: The two-pointer approach skips non-alphanumeric characters and compares characters from both ends, ensuring case-insensitive matching. If any mismatch occurs, the function returns `False`.

Programming Interview Guide: Linked Lists

1 Linked Lists

Linked list problems involve node manipulation, traversal, or cycle detection. This guide provides two example problems with detailed explanations and Python code implementations.

1.1 Example 1: Reverse Linked List

Problem Statement: Reverse a singly linked list.

Solution Approach:

- Use three pointers: `prev`, `current`, and `next`.
- Iterate through the list, reversing the `next` pointer of each node.

Time Complexity: $O(n)$, where n is the number of nodes.
Space Complexity: $O(1)$.

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def reverseList(head):
    prev = None
    current = head
    while current:
        next_node = current.next
        current.next = prev
        prev = current
        current = next_node
    return prev
```

Explanation: The algorithm reverses the list in-place by adjusting the `next` pointers. Each node points to the previous node, and the pointers are updated iteratively until the end.

1.2 Example 2: Merge Two Sorted Lists

Problem Statement: Merge two sorted linked lists into a new sorted linked list.

Solution Approach:

- Use a dummy node to simplify merging.
- Compare nodes from both lists, appending the smaller value.
- Attach the remaining nodes from either list.

Time Complexity: $O(n + m)$, where n and m are the lengths of the lists.
Space Complexity: $O(1)$.

```
def mergeTwoLists(list1, list2):
    dummy = ListNode(0)
    current = dummy
    while list1 and list2:
        if list1.val <= list2.val:
            current.next = list1
            list1 = list1.next
        else:
            current.next = list2
            list2 = list2.next
        current = current.next
    current.next = list1 if list1 else list2
    return dummy.next
```

Explanation: A dummy node simplifies the merging process. The algorithm compares nodes from both lists, appending the smaller one to the result and advancing the pointers accordingly.

Programming Interview Guide: Stacks, Queues, Deques

1 Stacks, Queues, Deques

Stacks (LIFO), queues (FIFO), and deques (double-ended queues) are fundamental data structures used in many algorithms. This guide provides two example problems with detailed explanations and Python code implementations.

1.1 Example 1: Valid Parentheses

Problem Statement: Given a string `s` containing just the characters '(', ')', '', '', '[' and ']', determine if the input string is valid. An input string is valid if open brackets are closed by the same type of brackets in the correct order.

Solution Approach:

- Use a stack to track opening brackets.
- For each closing bracket, check if it matches the top of the stack.
- If the stack is empty or there's a mismatch, return `False`.

Time Complexity: $O(n)$, where n is the string length.
Space Complexity: $O(n)$ for the stack.

```python
def isValid(s):
    stack = []
    brackets = {')': '(', '}': '{', ']': '['}
    for char in s:
        if char in brackets.values():
            stack.append(char)
        elif char in brackets:
            if not stack or stack.pop() != brackets[char]:
                return False
    return len(stack) == 0
```

Explanation: The algorithm uses a stack to store opening brackets. For each closing bracket, it checks if the stack's top matches the expected opening bracket. If the stack is empty or there's a mismatch, the string is invalid. Finally, it ensures the stack is empty.

Programming Interview Guide: Recursion & Backtracking

1 Recursion & Backtracking

Recursion and backtracking involve exploring all possible solutions by making choices and undoing them if necessary. This guide provides two example problems with detailed explanations and Python code implementations.

1.1 Example 1: Generate Parentheses

Problem Statement: Given `n` pairs of parentheses, generate all valid combinations of well-formed parentheses.

Solution Approach:

- Use backtracking to build valid combinations.
- Track the number of open and close parentheses.
- Add an open parenthesis if possible, or a close parenthesis if there are more open ones.

Time Complexity: $O(4^n/\sqrt{n})$ (Catalan number).
Space Complexity: $O(n)$ for the recursion stack.

```python
def generateParenthesis(n):
    result = []

    def backtrack(s, open_count, close_count):
        if len(s) == 2 * n:
            result.append(s)
            return
        if open_count < n:
            backtrack(s + '(', open_count + 1, close_count)
        if close_count < open_count:
            backtrack(s + ')', open_count, close_count + 1)

    backtrack('', 0, 0)
    return result
```

Explanation: The backtracking function builds strings by adding an open parenthesis if less than `n` are used, or a close parenthesis if there are more open ones. When the string length reaches $2n$, it is added to the result.

1.2 Example 2: Permutations

Problem Statement: Given an array `nums` of distinct integers, return all possible permutations.

Solution Approach:

- Use backtracking to generate all permutations.
- At each step, choose an unused number, add it to the current permutation, and recurse.
- Backtrack by removing the number and marking it unused.

Time Complexity: $O(n!)$, where n is the array length.
Space Complexity: $O(n)$ for the recursion stack.

```python
def permute(nums):
    result = []

    def backtrack(used, current):
        if len(current) == len(nums):
            result.append(current[:])
            return
        for i in range(len(nums)):
            if not used[i]:
                used[i] = True
                current.append(nums[i])
                backtrack(used, current)
                current.pop()
                used[i] = False

    backtrack([False] * len(nums), [])
    return result
```

Explanation: The backtracking function builds permutations by selecting each unused number, adding it to the current permutation, and recursing. After exploring, it backtracks by removing the number and marking it unused.

Programming Interview Guide: Dynamic Programming

1 Dynamic Programming

Dynamic programming (DP) problems involve breaking problems into overlapping subproblems. This guide provides two example problems with detailed explanations and Python code implementations.

1.1 Example 1: Fibonacci Number

Problem Statement: Given an integer `n`, return the `n`th Fibonacci number.

Solution Approach:

- Use a bottom-up DP approach with a small array.
- Store only the last two numbers to compute the next.

Time Complexity: $O(n)$.
Space Complexity: $O(1)$.

```python
def fib(n):
    if n <= 1:
        return n
    a, b = 0, 1
    for _ in range(2, n + 1):
        a, b = b, a + b
    return b
```

Explanation: The iterative approach uses two variables to store the previous two Fibonacci numbers, updating them in each iteration to compute the `n`th number efficiently.

1.2 Example 2: Longest Common Subsequence

Problem Statement: Given two strings `text1` and `text2`, return the length of their longest common subsequence.

Solution Approach:

- Use a 2D DP table to store lengths of common subsequences.

- Fill the table by comparing characters and using previous values.

Time Complexity: $O(m \cdot n)$, where m and n are the lengths of the strings.
Space Complexity: $O(m \cdot n)$ for the DP table.

```python
def longestCommonSubsequence(text1, text2):
    m, n = len(text1), len(text2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if text1[i-1] == text2[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])
    return dp[m][n]
```

Explanation: The DP table stores the lengths of the longest common subsequences for prefixes of the input strings. If characters match, the length increases by one; otherwise, the maximum of the previous values is taken.

Programming Interview Guide: Hash Tables / Hash Maps

1 Hash Tables / Hash Maps

Hash tables store key-value pairs for efficient lookup and manipulation. This guide provides two example problems with detailed explanations and Python code implementations.

1.1 Example 1: Two Sum

Problem Statement: Given an array of integers `nums` and an integer `target`, find two numbers such that they add up to `target`. Return their indices. Assume exactly one solution exists.

Solution Approach:

- Use a hash map to store numbers and their indices.
- Iterate through the array, checking if `target - nums[i]` exists in the hash map.
- If found, return the current index and the stored index.

Time Complexity: $O(n)$, where n is the array length.
Space Complexity: $O(n)$ for the hash map.

```
def twoSum(nums, target):
    seen = {}
    for i, num in enumerate(nums):
        complement = target - num
        if complement in seen:
            return [seen[complement], i]
        seen[num] = i
    return []
```

Explanation: The function uses a dictionary to store each number and its index. For each number, it calculates the complement (`target - num`) and checks if it exists in the dictionary. If it does, it returns the indices.

1.2 Example 2: Group Anagrams

Problem Statement: Given an array of strings `strs`, group the anagrams together. Return the grouped anagrams as a list of lists.

Solution Approach:

- Use a hash map where the key is the sorted string and the value is a list of anagrams.
- Iterate through the strings, sorting each to create the key.
- Group strings with the same sorted key together.

Time Complexity: $O(n \cdot k \cdot \log k)$, where n is the number of strings and k is the maximum string length.
Space Complexity: $O(n \cdot k)$ for the hash map.

```
def groupAnagrams(strs):
    anagrams = {}
    for s in strs:
        sorted_s = ''.join(sorted(s))
        if sorted_s not in anagrams:
            anagrams[sorted_s] = []
        anagrams[sorted_s].append(s)
    return list(anagrams.values())
```

Explanation: Each string is sorted to create a key for the hash map. Strings that are anagrams share the same sorted key and are grouped together in the hash map's value list. The result is the list of grouped anagrams.

Programming Interview Guide: Heaps / Priority Queues

1 Heaps / Priority Queues

Heaps are used to maintain a collection of elements with efficient access to the minimum or maximum. This guide provides two example problems with detailed explanations and Python code implementations using the `heapq` module.

1.1 Example 1: Kth Largest Element in an Array

Problem Statement: Given an integer array `nums` and an integer `k`, return the `k`th largest element in the array.

Solution Approach:

- Use a min-heap of size `k`.
- Push elements to the heap, popping the smallest if the heap size exceeds `k`.
- The heap's root is the `k`th largest element.

Time Complexity: $O(n \cdot \log k)$, where n is the array length.
Space Complexity: $O(k)$ for the heap.

```
import heapq

def findKthLargest(nums, k):
    heap = []
    for num in nums:
        heapq.heappush(heap, num)
        if len(heap) > k:
            heapq.heappop(heap)
    return heap[0]
```

Explanation: The min-heap maintains the `k` largest elements. For each element, it is pushed to the heap, and if the heap size exceeds `k`, the smallest element is removed. The root of the heap is the `k`th largest element.

1.2 Example 2: Merge k Sorted Lists

Problem Statement: Merge `k` sorted linked lists and return the merged sorted list.

Solution Approach:

- Use a min-heap to store the current node of each list.
- Pop the smallest node, add it to the result, and push the next node from the same list.
- Continue until the heap is empty.

Time Complexity: $O(n \cdot \log k)$, where n is the total number of nodes and k is the number of lists.
Space Complexity: $O(k)$ for the heap.

```python
import heapq

class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def mergeKLists(lists):
    heap = []
    for i, node in enumerate(lists):
        if node:
            heapq.heappush(heap, (node.val, i, node))

    dummy = ListNode(0)
    current = dummy

    while heap:
        val, i, node = heapq.heappop(heap)
        current.next = node
        current = current.next
        if node.next:
            heapq.heappush(heap, (node.next.val, i, node.next))

    return dummy.next
```

Explanation: The min-heap stores nodes from each list, prioritized by their values. The smallest node is popped, added to the result, and its next node (if any) is pushed to the heap. A tuple with the list index ensures proper handling of equal values.

Tree Guide for SWE Interviews

Generated by ChatGPT

July 6, 2025

Contents

1 Trees

1.1 Example 1: Maximum Depth of Binary Tree

- **Problem Statement:** Given the root of a binary tree, return its maximum depth.
- **Solution Approach:** Use recursive depth-first search (DFS). Return max depth of left and right subtrees plus one.
- **Time Complexity:** $O(n)$, where n is the number of nodes.
- **Space Complexity:** $O(h)$, where h is the height of the tree.
- **Code:**

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def maxDepth(root):
    if not root:
        return 0
    left_depth = maxDepth(root.left)
    right_depth = maxDepth(root.right)
    return max(left_depth, right_depth) + 1
```

- **Explanation:** The recursive function calculates the depth by taking the max of left and right subtree depths and adding one.

1.2 Example 2: Validate Binary Search Tree

- **Problem Statement:** Given a binary tree, determine if it is a valid binary search tree (BST).
- **Solution Approach:** Use recursive DFS with a valid range for each node.
- **Time Complexity:** $O(n)$, where n is the number of nodes.
- **Space Complexity:** $O(h)$, where h is the height of the tree.
- **Code:**

```
def isValidBST(root):
    def validate(node, min_val, max_val):
        if not node:
            return True
        if node.val <= min_val or node.val >= max_val:
            return False
        return validate(node.left, min_val, node.val) and \
               validate(node.right, node.val, max_val)
    return validate(root, float('-inf'), float('inf'))
```

- **Explanation:** The function ensures that each node's value is within a valid range, adjusting the range recursively.

Programming Interview Guide: Graphs (BFS, DFS, Topological Sort)

1 Graphs (BFS, DFS, Topological Sort)

Graph problems involve traversal (BFS, DFS) or ordering (Topological Sort). This guide provides two example problems with detailed explanations and Python code implementations.

1.1 Example 1: Number of Islands

Problem Statement: Given a 2D grid map of '1's (land) and '0's (water), count the number of islands. An island is surrounded by water and formed by connecting adjacent lands.

Solution Approach:

- Use DFS to explore each island.
- When a '1' is found, mark it as visited and explore all adjacent lands.
- Increment the island count for each new island found.

Time Complexity: $O(m \cdot n)$, where m and n are the grid dimensions.
Space Complexity: $O(m \cdot n)$ for the recursion stack in worst case.

```python
def numIslands(grid):
    if not grid:
        return 0
    rows, cols = len(grid), len(grid[0])
    islands = 0

    def dfs(i, j):
        if i < 0 or i >= rows or j < 0 or j >= cols or grid[i][j]
            != "1":
            return
        grid[i][j] = "0"  # Mark as visited
        dfs(i+1, j)
        dfs(i-1, j)
        dfs(i, j+1)
        dfs(i, j-1)

    for i in range(rows):
```

```
        for j in range(cols):
            if grid[i][j] == "1":
                dfs(i, j)
                islands += 1
    return islands
```

Explanation: The DFS function marks connected '1's as visited by setting them to '0' and recursively explores all four directions. Each unvisited '1' triggers a new DFS, incrementing the island count.

1.2 Example 2: Course Schedule II

Problem Statement: Given `numCourses` and a list of prerequisites, return the ordering of courses to take to finish all courses (topological sort). If impossible, return an empty array.

Solution Approach:

- Build an adjacency list and track in-degrees.
- Use topological sort with a queue, starting with courses having no prerequisites.
- Process each course, reducing in-degrees and adding courses with zero in-degrees to the queue.

Time Complexity: $O(V + E)$, where V is the number of courses and E is the number of prerequisites.
Space Complexity: $O(V + E)$ for the adjacency list and queue.

```
from collections import defaultdict, deque

def findOrder(numCourses, prerequisites):
    graph = defaultdict(list)
    in_degree = [0] * numCourses
    for dest, src in prerequisites:
        graph[src].append(dest)
        in_degree[dest] += 1

    queue = deque([i for i in range(numCourses) if in_degree[i] ==
       0])
    order = []

    while queue:
        course = queue.popleft()
        order.append(course)
        for neighbor in graph[course]:
            in_degree[neighbor] -= 1
            if in_degree[neighbor] == 0:
                queue.append(neighbor)

    return order if len(order) == numCourses else []
```

Explanation: The algorithm builds a directed graph and tracks in-degrees. Courses with no prerequisites are processed first, and their neighbors' in-degrees are decremented. If all courses are processed, a valid order is returned; otherwise, a cycle exists, so an empty list is returned.

Programming Interview Guide: Greedy Algorithms

1 Greedy Algorithms

Greedy algorithms make locally optimal choices to achieve a global optimum. This guide provides two example problems with detailed explanations and Python code implementations.

1.1 Example 1: Maximum Meetings

Problem Statement: Given arrays `start` and `end` representing meeting start and end times, find the maximum number of non-overlapping meetings that can be attended.

Solution Approach:

- Sort meetings by end time.
- Greedily select the meeting that ends earliest and is compatible with the current schedule.
- Update the last selected meeting's end time.

Time Complexity: $O(n \cdot \log n)$ due to sorting.
Space Complexity: $O(1)$ excluding input storage.

```python
def maxMeetings(start, end):
    meetings = sorted(zip(start, end), key=lambda x: x[1])
    count = 0
    last_end = -1
    for s, e in meetings:
        if s > last_end:
            count += 1
            last_end = e
    return count
```

Explanation: Meetings are sorted by end time to prioritize those that free up the schedule soonest. The algorithm selects a meeting if its start time is after the last selected meeting's end time, incrementing the count and updating the end time.

1.2 Example 2: Minimum Number of Platforms

Problem Statement: Given arrays `arr` and `dep` representing train arrival and departure times, find the minimum number of platforms needed so that no train waits.

Solution Approach:

- Sort arrival and departure times separately.
- Use two pointers to track arrivals and departures, incrementing platform count for arrivals and decrementing for departures.
- Track the maximum platforms needed at any time.

Time Complexity: $O(n \cdot \log n)$ due to sorting.
Space Complexity: $O(1)$.

```
def minPlatforms(arr, dep):
    arr.sort()
    dep.sort()
    platforms = 0
    max_platforms = 0
    i, j = 0, 0
    while i < len(arr):
        if i < len(arr) and (j >= len(dep) or arr[i] <= dep[j]):
            platforms += 1
            max_platforms = max(max_platforms, platforms)
            i += 1
        else:
            platforms -= 1
            j += 1
    return max_platforms
```

Explanation: The algorithm processes events in chronological order by sorting arrivals and departures. When an arrival occurs, a platform is needed; when a departure occurs, a platform is freed. The maximum number of platforms needed at any point is tracked.

Programming Interview Guide: Sliding Window / Two Pointers

1 Sliding Window / Two Pointers

Sliding window and two-pointer techniques optimize problems involving arrays or strings by maintaining a window or pointers. This guide provides two example problems with detailed explanations and Python code implementations.

1.1 Example 1: Longest Substring Without Repeating Characters

Problem Statement: Given a string `s`, find the length of the longest substring without repeating characters.

Solution Approach:

- Use a sliding window with a hash map to track character positions.
- Move the right pointer, updating the window start if a repeat is found.
- Track the maximum length of the window.

Time Complexity: $O(n)$, where n is the string length.
Space Complexity: $O(\min(m, n))$, where m is the size of the character set.

```python
def lengthOfLongestSubstring(s):
    seen = {}
    max_length = 0
    start = 0
    for end in range(len(s)):
        if s[end] in seen and seen[s[end]] >= start:
            start = seen[s[end]] + 1
        else:
            max_length = max(max_length, end - start + 1)
        seen[s[end]] = end
    return max_length
```

Explanation: The sliding window tracks the longest substring without repeats. When a repeating character is found, the window's start is moved past the previous occurrence. The hash map stores the latest index of each character.

1.2 Example 2: Container With Most Water

Problem Statement: Given an array `height` representing heights of vertical lines, find two lines that form a container with the most water.

Solution Approach:

- Use two pointers starting from both ends.
- Compute the area between the pointers and move the pointer at the shorter line inward.
- Track the maximum area found.

Time Complexity: $O(n)$, where n is the array length.
Space Complexity: $O(1)$.

```python
def maxArea(height):
    max_water = 0
    left, right = 0, len(height) - 1
    while left < right:
        width = right - left
        h = min(height[left], height[right])
        max_water = max(max_water, width * h)
        if height[left] < height[right]:
            left += 1
        else:
            right -= 1
    return max_water
```

Explanation: The two pointers start at the array's ends. The area is calculated as the minimum height times the width. Moving the pointer at the shorter line maximizes potential area, as a taller line might yield a larger area despite a smaller width.

Programming Interview Guide: Bit Manipulation

1 Bit Manipulation

Bit manipulation involves using bitwise operations to solve problems efficiently. This guide provides two example problems with detailed explanations and Python code implementations.

1.1 Example 1: Single Number

Problem Statement: Given a non-empty array of integers `nums`, every element appears twice except for one. Find that single number.

Solution Approach:

- Use XOR operation, as XOR of a number with itself is 0, and XOR with 0 is the number itself.
- XOR all numbers in the array to find the single number.

Time Complexity: $O(n)$, where n is the array length.
Space Complexity: $O(1)$.

```python
def singleNumber(nums):
    result = 0
    for num in nums:
        result ^= num
    return result
```

Explanation: XORing all numbers cancels out pairs (since $a \oplus a = 0$), leaving the single number. The operation is efficient and requires no extra space.

1.2 Example 2: Number of 1 Bits

Problem Statement: Given an unsigned integer `n`, return the number of '1' bits (Hamming weight).

Solution Approach:

- Use bitwise AND and right shift to check each bit.
- Count bits that are 1 until the number becomes 0.

Time Complexity: $O(1)$ for a 32-bit integer.
Space Complexity: $O(1)$.

```
def hammingWeight(n):
    count = 0
    while n:
        count += n & 1
        n >>= 1
    return count
```

Explanation: The algorithm checks the least significant bit using `n & 1`. If it is 1, the count is incremented. The number is right-shifted to check the next bit, continuing until `n` is 0.