Solutions to DSA Questions 111-141 (Graphs, DP, Bit Manipulation) For 1-2 Years Experience Roles at EPAM

Compiled on September 27, 2025

Introduction

This document provides detailed solutions for 31 Data Structures and Algorithms (DSA) problems (questions 111 to 141) from the Graphs, Dynamic Programming, and Bit Manipulation categories, tailored for candidates with 1-2 years of experience preparing for roles at EPAM Systems. Each problem includes a problem statement, dry run with test cases, algorithm, and a Python solution, formatted for clarity. Dynamic programming problems include both memoization and tabulation approaches where applicable.

Contents

1 Pacific Atlantic Water Flow

1.1 Problem Statement

Given an m x n matrix of heights, find all cells where water can flow to both Pacific and Atlantic oceans.

1.2 Dry Run on Test Cases

- **Test Case 1**: heights = [[1,2,2,3,5],[3,2,3,4,4],[2,4,5,3,1],[6,7,1,4,5],[5,1,1,2,4]] → Output: [[0,4],[1,3],[1,4],[2,2],[3,0],[3,1],[4,0]]
- **Test Case 2**: heights = [[1]] → Output: [[0,0]]
- **Test Case 3**: heights = [[1,2],[2,1]] → Output: [[0,0],[0,1],[1,0],[1,1]]
- **Test Case 4**: heights = [] → Output: []

1.3 Algorithm

1. Use DFS from Pacific (top/left) and Atlantic (bottom/right) borders.

2. Mark reachable cells for each ocean.
3. Return cells reachable by both.

Time Complexity: $O(m \cdot n)$ **Space Complexity**: $O(m \cdot n)$

1.4 Python Solution

```python
def pacific_atlantic(heights):
    if not heights or not heights[0]:
        return []

    rows, cols = len(heights), len(heights[0])
    pacific = set()
    atlantic = set()

    def dfs(i, j, visited, prev_height):
        if i < 0 or i >= rows or j < 0 or j >= cols or (i, j) in
           visited or heights[i][j] < prev_height:
            return
        visited.add((i, j))
        for di, dj in [(1,0), (-1,0), (0,1), (0,-1)]:
            dfs(i + di, j + dj, visited, heights[i][j])

    # Pacific: top and left borders
    for j in range(cols):
        dfs(0, j, pacific, float('-inf'))
    for i in range(rows):
        dfs(i, 0, pacific, float('-inf'))

    # Atlantic: bottom and right borders
    for j in range(cols):
        dfs(rows-1, j, atlantic, float('-inf'))
    for i in range(rows):
        dfs(i, cols-1, atlantic, float('-inf'))

    return list(pacific & atlantic)
```

2 Surrounded Regions

2.1 Problem Statement

Given a 2D board of 'X' and 'O', flip all 'O's surrounded by 'X' to 'X'.

2.2 Dry Run on Test Cases

- **Test Case 1**: board = [["X","X","X","X"],["X","O","O","X"],["X","X","O","X"],["X","O","X","X"]] → Output: [["X","X","X","X"],["X","X","X","X"],["X","X","X","X"],["X","O","X","X"]]
- **Test Case 2**: board = [["O"]] → Output: [["O"]]

- **Test Case 3**: board = [] → Output: []
- **Test Case 4**: board = [["X"]] → Output: [["X"]]

2.3 Algorithm

1. Use DFS to mark 'O's connected to border as safe.
2. Flip unmarked 'O's to 'X'.
3. Restore safe 'O's.

Time Complexity: $O(m \cdot n)$ **Space Complexity**: $O(m \cdot n)$

2.4 Python Solution

```python
def solve(board):
    if not board or not board[0]:
        return

    rows, cols = len(board), len(board[0])

    def dfs(i, j):
        if i < 0 or i >= rows or j < 0 or j >= cols or board[i][j
            ] != 'O':
            return
        board[i][j] = '#'
        for di, dj in [(1,0), (-1,0), (0,1), (0,-1)]:
            dfs(i + di, j + dj)

    # Mark border-connected 'O's
    for i in range(rows):
        if board[i][0] == 'O':
            dfs(i, 0)
        if board[i][cols-1] == 'O':
            dfs(i, cols-1)
    for j in range(cols):
        if board[0][j] == 'O':
            dfs(0, j)
        if board[rows-1][j] == 'O':
            dfs(rows-1, j)

    # Flip 'O' to 'X', '#' to 'O'
    for i in range(rows):
        for j in range(cols):
            if board[i][j] == 'O':
                board[i][j] = 'X'
            elif board[i][j] == '#':
                board[i][j] = 'O'
```

3 Graph Valid Tree

3.1 Problem Statement

Given n nodes and edges, check if they form a valid tree (connected, no cycles).

3.2 Dry Run on Test Cases

- **Test Case 1**: n = 5, edges = [[0,1],[0,2],[0,3],[1,4]] → Output: True
- **Test Case 2**: n = 5, edges = [[0,1],[1,2],[2,3],[1,3],[1,4]] → Output: False
- **Test Case 3**: n = 1, edges = [] → Output: True
- **Test Case 4**: n = 2, edges = [] → Output: False

3.3 Algorithm

1. Use DFS to detect cycles and check connectivity.
2. Build adjacency list.
3. Ensure all nodes visited and no cycles.

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

3.4 Python Solution

```python
from collections import defaultdict

def valid_tree(n, edges):
    if len(edges) != n - 1:
        return False

    graph = defaultdict(list)
    for u, v in edges:
        graph[u].append(v)
        graph[v].append(u)

    visited = set()

    def dfs(node, parent):
        visited.add(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
                if not dfs(neighbor, node):
                    return False
            elif neighbor != parent:
                return False
        return True

    return dfs(0, -1) and len(visited) == n
```

4 Minimum Spanning Tree (Kruskal's Algorithm)

4.1 Problem Statement

Given a weighted undirected graph, find the minimum spanning tree weight using Kruskal's algorithm.

4.2 Dry Run on Test Cases

- **Test Case 1**: n = 4, edges = [[0,1,1],[0,2,2],[1,2,5],[1,3,1],[2,3,4]] → Output: 4
- **Test Case 2**: n = 1, edges = [] → Output: 0
- **Test Case 3**: n = 2, edges = [[0,1,1]] → Output: 1
- **Test Case 4**: n = 3, edges = [[0,1,2],[1,2,2]] → Output: 4

4.3 Algorithm

1. Sort edges by weight.
2. Use Union-Find to avoid cycles.
3. Add edges to MST if they connect different components.

Time Complexity: $O(E \log E)$ **Space Complexity**: $O(V)$

4.4 Python Solution

```python
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        px, py = self.find(x), self.find(y)
        if px == py:
            return False
        if self.rank[px] < self.rank[py]:
            px, py = py, px
        self.parent[py] = px
        if self.rank[px] == self.rank[py]:
            self.rank[px] += 1
        return True

def minimum_spanning_tree(n, edges):
    edges.sort(key=lambda x: x[2])
    uf = UnionFind(n)
    total_weight = 0
    edges_used = 0

    for u, v, weight in edges:
        if uf.union(u, v):
            total_weight += weight
            edges_used += 1

    return total_weight if edges_used == n - 1 else -1
```

5 Dijkstra's Algorithm

5.1 Problem Statement

Given a weighted directed graph and source, find shortest paths to all vertices.

5.2 Dry Run on Test Cases

- **Test Case 1**: n = 4, edges = [[0,1,1],[0,2,4],[1,2,2],[1,3,6],[2,3,3]], source = 0 $\rightarrow$ Output: [0,1,3,6]
- **Test Case 2**: n = 1, edges = [], source = 0 $\rightarrow$ Output: [0]
- **Test Case 3**: n = 2, edges = [[0,1,1]], source = 0 $\rightarrow$ Output: [0,1]
- **Test Case 4**: n = 3, edges = [[0,1,2],[1,2,2]], source = 0 $\rightarrow$ Output: [0,2,4]

5.3 Algorithm

1. Use min-heap to store (distance, node).
2. Update distances to neighbors if shorter path found.
3. Return distance array.

Time Complexity: $O((V + E)\log V)$ **Space Complexity**: $O(V + E)$

5.4 Python Solution

```python
import heapq
from collections import defaultdict

def dijkstra(n, edges, source):
    graph = defaultdict(list)
    for u, v, w in edges:
        graph[u].append((v, w))

    distances = [float('inf')] * n
    distances[source] = 0
    heap = [(0, source)]

    while heap:
        dist, node = heapq.heappop(heap)
        if dist > distances[node]:
            continue
        for neighbor, weight in graph[node]:
            if dist + weight < distances[neighbor]:
                distances[neighbor] = dist + weight
                heapq.heappush(heap, (dist + weight,
                   neighbor))

    return [d if d != float('inf') else -1 for d in
       distances]
```

6 Climbing Stairs

6.1 Problem Statement

Given n stairs, find the number of ways to climb (1 or 2 steps at a time).

6.2 Dry Run on Test Cases

* **Test Case 1**: n = 2 → Output: 2 ([1,1], [2])
* **Test Case 2**: n = 3 → Output: 3 ([1,1,1], [1,2], [2,1])
* **Test Case 3**: n = 1 → Output: 1 ([1])
* **Test Case 4**: n = 4 → Output: 5

6.3 Algorithm (Memoization)

1. Use recursive function with memoization.
2. Base cases: n=0 (1 way), n<0 (0 ways).
3. Return sum of ways for n-1 and n-2.

6.4 Algorithm (Tabulation)

1. Use DP array where dp[i] = ways to climb i stairs.
2. Initialize dp[0]=1, dp[1]=1.
3. dp[i] = dp[i-1] + dp[i-2] for i from 2 to n.

Time Complexity: $O(n)$ **Space Complexity**: $O(n)$ (memoization), $O(1)$ (tabulation)

6.5 Python Solution (Memoization)

```python
def climb_stairs_memo(n, memo=None):
    if memo is None:
        memo = {}
    if n in memo:
        return memo[n]
    if n <= 0:
        return 1 if n == 0 else 0
    memo[n] = climb_stairs_memo(n-1, memo) +
       climb_stairs_memo(n-2, memo)
    return memo[n]
```

6.6 Python Solution (Tabulation)

```python
def climb_stairs(n):
    if n <= 1:
        return 1
    dp = [0] * (n + 1)
    dp[0] = dp[1] = 1
    for i in range(2, n + 1):
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]
```

7 Min Cost Climbing Stairs

7.1 Problem Statement

Given a cost array for climbing stairs, find minimum cost to reach the top.

7.2 Dry Run on Test Cases

* **Test Case 1**: cost = [10,15,20] $\rightarrow$ Output: 15
* **Test Case 2**: cost = [1,100,1,1,1,100,1,1,100,1] $\rightarrow$ Output: 6
* **Test Case 3**: cost = [0,1] $\rightarrow$ Output: 0
* **Test Case 4**: cost = [1] $\rightarrow$ Output: 0

7.3 Algorithm (Memoization)

1. Recurse with memoization for min cost from index i.
2. Base case: i >= len(cost) returns 0.
3. Min cost = cost[i] + min(cost(i+1), cost(i+2)).

7.4 Algorithm (Tabulation)

1. dp[i] = min cost to reach top from i.
2. dp[n] = 0, dp[n-1] = cost[n-1].
3. dp[i] = cost[i] + min(dp[i+1], dp[i+2]).

Time Complexity: $O(n)$ **Space Complexity**: $O(n)$ (memoization), $O(1)$ (tabulation)

7.5 Python Solution (Memoization)

```python
def min_cost_climbing_stairs_memo(cost):
    memo = {}
    def dp(i):
        if i >= len(cost):
            return 0
        if i in memo:
            return memo[i]
        memo[i] = cost[i] + min(dp(i+1), dp(i+2))
        return memo[i]
    return min(dp(0), dp(1))
```

7.6 Python Solution (Tabulation)

```python
def min_cost_climbing_stairs(cost):
    n = len(cost)
    dp = [0] * (n + 1)
    for i in range(n-1, -1, -1):
        dp[i] = cost[i] + min(dp[i+1], dp[i+2])
    return min(dp[0], dp[1])
```

8 House Robber

8.1 Problem Statement

Given an array of house values, find max amount to rob without robbing adjacent houses.

8.2 Dry Run on Test Cases

* **Test Case 1**: nums = [1,2,3,1] → Output: 4 (1+3)
* **Test Case 2**: nums = [2,7,9,3,1] → Output: 12 (2+9+1)
* **Test Case 3**: nums = [1] → Output: 1
* **Test Case 4**: nums = [] → Output: 0

8.3 Algorithm (Memoization)

1. Recurse with memoization for max from index i.
2. Base case: i >= len(nums) returns 0.
3. Max = max(nums[i] + dp(i+2), dp(i+1)).

8.4 Algorithm (Tabulation)

1. dp[i] = max amount from index i to end.
2. dp[n] = 0, dp[n-1] = nums[n-1].
3. dp[i] = max(nums[i] + dp[i+2], dp[i+1]).

Time Complexity: $O(n)$ **Space Complexity**: $O(n)$ (memoization), $O(1)$ (tabulation)

8.5 Python Solution (Memoization)

```python
def rob_memo(nums):
    memo = {}
    def dp(i):
        if i >= len(nums):
            return 0
        if i in memo:
            return memo[i]
        memo[i] = max(nums[i] + dp(i+2), dp(i+1))
        return memo[i]
    return dp(0)
```

8.6 Python Solution (Tabulation)

```python
def rob(nums):
    if not nums:
        return 0
    if len(nums) == 1:
        return nums[0]

    dp = [0] * (len(nums) + 1)
    dp[-2] = nums[-1]
    for i in range(len(nums)-2, -1, -1):
        dp[i] = max(nums[i] + dp[i+2], dp[i+1])
    return dp[0]
```

9 House Robber II

9.1 Problem Statement

Given a circular array of house values, find max amount to rob without robbing adjacent houses.

9.2 Dry Run on Test Cases

- **Test Case 1**: nums = [2,3,2] → Output: 3
- **Test Case 2**: nums = [1,2,3,1] → Output: 4
- **Test Case 3**: nums = [1] → Output: 1
- **Test Case 4**: nums = [] → Output: 0

9.3 Algorithm (Memoization)

1. Solve House Robber for nums[0:n-1] and nums[1:n].
2. Return max of both results.

9.4 Algorithm (Tabulation)

1. Use tabulation for two cases: exclude first and exclude last house.
2. dp[i] = max(nums[i] + dp[i+2], dp[i+1]).

Time Complexity: $O(n)$ **Space Complexity**: $O(n)$ (memoization), $O(1)$ (tabulation)

9.5 Python Solution (Tabulation)

```python
def rob_circular(nums):
    if not nums:
        return 0
    if len(nums) == 1:
        return nums[0]

    def rob_linear(arr):
        if not arr:
            return 0
        if len(arr) == 1:
            return arr[0]
        dp = [0] * (len(arr) + 1)
        dp[-2] = arr[-1]
        for i in range(len(arr)-2, -1, -1):
            dp[i] = max(arr[i] + dp[i+2], dp[i+1])
        return dp[0]

    return max(rob_linear(nums[:-1]), rob_linear(
       nums[1:]))
```

10 Coin Change

10.1 Problem Statement

Given coins and an amount, find minimum coins needed to make the amount.

10.2 Dry Run on Test Cases

- **Test Case 1**: coins = [1,2,5], amount = 11 → Output: 3 (5+5+1)
- **Test Case 2**: coins = [2], amount = 3 → Output: -1
- **Test Case 3**: coins = [1], amount = 0 → Output: 0
- **Test Case 4**: coins = [1,2], amount = 1 → Output: 1

10.3 Algorithm (Memoization)

1. Recurse with memoization for min coins for amount.
2. Base case: amount=0 returns 0, amount<0 returns inf.
3. Try each coin and take minimum.

10.4 Algorithm (Tabulation)

1. dp[i] = min coins for amount i.
2. Initialize dp[0]=0, others=inf.
3. For each amount, try each coin, update dp.

Time Complexity: $O(amount \cdot n)$ **Space Complexity**: $O(amount)$

10.5 Python Solution (Memoization)

```python
def coin_change_memo(coins, amount):
    memo = {}
    def dp(amt):
        if amt in memo:
            return memo[amt]
        if amt == 0:
            return 0
        if amt < 0:
            return float('inf')
        min_coins = float('inf')
        for coin in coins:
            min_coins = min(min_coins, dp(amt -
                coin) + 1)
        memo[amt] = min_coins
        return memo[amt]
    result = dp(amount)
    return result if result != float('inf') else -1
```

10.6 Python Solution (Tabulation)

```
def coin_change(coins, amount):
    dp = [float('inf')] * (amount + 1)
    dp[0] = 0
    for i in range(1, amount + 1):
        for coin in coins:
            if i >= coin:
                dp[i] = min(dp[i], dp[i - coin] +
                    1)
    return dp[amount] if dp[amount] != float('inf')
        else -1
```

11 Coin Change II

11.1 Problem Statement

Given coins and an amount, find number of ways to make the amount.

11.2 Dry Run on Test Cases

- **Test Case 1**: amount = 5, coins = [1,2,5] → Output: 4 ([1,1,1,1,1], [1,1,1,2], [1,2,2], [5])
- **Test Case 2**: amount = 3, coins = [2] → Output: 0
- **Test Case 3**: amount = 10, coins = [10] → Output: 1
- **Test Case 4**: amount = 0, coins = [1] → Output: 1

11.3 Algorithm (Memoization)

1. Recurse with memoization for ways with index i and amount.
2. Base case: amount=0 returns 1, amount<0 or i>=len(coins) returns 0.
3. Sum ways by including or excluding current coin.

11.4 Algorithm (Tabulation)

1. dp[i] = ways to make amount i.
2. Initialize dp[0]=1.

3. For each coin, update dp[j] += dp[j-coin].

Time Complexity: $O(amount \cdot n)$ **Space Complexity**: $O(amount)$

11.5 Python Solution (Memoization)

```python
def change_memo(amount, coins):
    memo = {}
    def dp(i, amt):
        if (i, amt) in memo:
            return memo[(i, amt)]
        if amt == 0:
            return 1
        if amt < 0 or i >= len(coins):
            return 0
        memo[(i, amt)] = dp(i, amt - coins[i]) + dp
           (i + 1, amt)
        return memo[(i, amt)]
    return dp(0, amount)
```

11.6 Python Solution (Tabulation)

```python
def change(amount, coins):
    dp = [0] * (amount + 1)
    dp[0] = 1
    for coin in coins:
        for i in range(coin, amount + 1):
            dp[i] += dp[i - coin]
    return dp[amount]
```

12 Longest Increasing Subsequence

12.1 Problem Statement

Given an array, find the length of the longest increasing subsequence.

12.2 Dry Run on Test Cases

- **Test Case 1**: nums = [10,9,2,5,3,7,101,18] → Output: 4 ([2,3,7,101])
- **Test Case 2**: nums = [0,1,0,3,2,3] → Output: 4
- **Test Case 3**: nums = [7,7,7,7] → Output: 1
- **Test Case 4**: nums = [] → Output: 0

12.3 Algorithm (Memoization)

1. Recurse with memoization for LIS ending at i with prev value.

2. Base case: i >= len(nums) returns 0.

3. Include nums[i] if greater than prev, else skip.

12.4 Algorithm (Tabulation)

1. dp[i] = length of LIS ending at i.

2. Initialize dp[i]=1.

3. For each i, check j<i where nums[j]<nums[i], update dp[i].

Time Complexity: $O(n^2)$ **Space Complexity**: $O(n)$

12.5 Python Solution (Memoization)

```python
def length_of_lis_memo(nums):
    memo = {}
    def dp(i, prev):
        if i >= len(nums):
            return 0
        if (i, prev) in memo:
            return memo[(i, prev)]
        take = 0
        if prev == -1 or nums[i] > nums[prev]:
            take = 1 + dp(i + 1, i)
        skip = dp(i + 1, prev)
        memo[(i, prev)] = max(take, skip)
        return memo[(i, prev)]
    return dp(0, -1)
```

12.6 Python Solution (Tabulation)

```python
def length_of_lis(nums):
    if not nums:
        return 0
    dp = [1] * len(nums)
    for i in range(1, len(nums)):
        for j in range(i):
            if nums[i] > nums[j]:
                dp[i] = max(dp[i], dp[j] + 1)
    return max(dp)
```

13 Longest Common Subsequence

13.1 Problem Statement

Given two strings, find the length of their longest common subsequence.

13.2 Dry Run on Test Cases

- **Test Case 1**: text1 = "abcde", text2 = "ace" → Output: 3 ("ace")
- **Test Case 2**: text1 = "abc", text2 = "abc" → Output: 3
- **Test Case 3**: text1 = "abc", text2 = "def" → Output: 0
- **Test Case 4**: text1 = "", text2 = "a" → Output: 0

13.3 Algorithm (Memoization)

1. Recurse with memoization for LCS of prefixes i, j.
2. If characters match, add 1 and recurse.
3. Else, take max of skipping one character.

13.4 Algorithm (Tabulation)

1. dp[i][j] = LCS length for prefixes i, j.
2. If characters match, dp[i][j] = dp[i-1][j-1] + 1.
3. Else, dp[i][j] = max(dp[i-1][j], dp[i][j-1]).

Time Complexity: $O(m \cdot n)$ **Space Complexity**: $O(m \cdot n)$

13.5 Python Solution (Memoization)

```python
def longest_common_subsequence_memo(text1, text2):
    memo = {}
    def dp(i, j):
        if i >= len(text1) or j >= len(text2):
            return 0
        if (i, j) in memo:
            return memo[(i, j)]
        if text1[i] == text2[j]:
            memo[(i, j)] = 1 + dp(i + 1, j + 1)
        else:
            memo[(i, j)] = max(dp(i + 1, j), dp(i,
               j + 1))
        return memo[(i, j)]
    return dp(0, 0)
```

13.6 Python Solution (Tabulation)

```python
def longest_common_subsequence(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]
\end Prefeitura

% Problem 124
\section{Edit Distance}
\subsection{Problem Statement}
Given two strings, find minimum operations (insert, delete, replace) to convert one to the other.

\subsection{Dry Run on Test Cases}
\begin{itemize}
    \item \textbf{Test Case 1}: word1 = "horse", word2 = "ros" $\rightarrow$ Output: 3
    \item \textbf{Test Case 2}: word1 = "intention", word2 = "execution" $\rightarrow$ Output: 5
    \item \textbf{Test Case 3}: word1 = "", word2 = "a" $\rightarrow$ Output: 1
    \item \textbf{Test Case 4}: word1 = "abc", word2 = "abc" $\rightarrow$ Output: 0
\end{itemize]

\subsection{Algorithm (Memoization)}
\begin{enumerate}
    \item Recurse with memoization for min operations for prefixes i, j.
    \item If characters match, no operation needed.
    \item Else, try insert, delete, replace, take minimum.
\end{enumerate}
\subsection{Algorithm (Tabulation)}
\begin{enumerate}
    \item dp[i][j] = min operations for prefixes i, j.
    \item Initialize dp[0][j] = j, dp[i][0] = i.
    \item If match, dp[i][j] = dp[i-1][j-1].
    \item Else, dp[i][j] = min(insert, delete, replace) + 1.
\end{enumerate}
\textbf{Time Complexity}: $O(m \cdot n)$ \quad \textbf{Space Complexity}: $O(m \cdot n)$

\subsection{Python Solution (Memoization)}
\begin{lstlisting}
def min_distance_memo(word1, word2):
    memo = {}
    def dp(i, j):
        if i >= len(word1):
            return len(word2) - j
        if j >= len(word2):
            return len(word1) - i
        if (i, j) in memo:
            return memo[(i, j)]
        if word1[i] == word2[j]:
            memo[(i, j)] = dp(i + 1, j + 1)
        else:
            memo[(i, j)] = min(dp(i, j + 1), dp(i +
                1, j), dp(i + 1, j + 1)) + 1
        return memo[(i, j)]
    return dp(0, 0)
```

13.7 Python Solution (Tabulation)

```
def min_distance(word1, word2):
    m, n = len(word1), len(word2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]
    for j in range(n + 1):
        dp[0][j] = j
    for i in range(m + 1):
        dp[i][0] = i
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if word1[i-1] == word2[j-1]:
                dp[i][j] = dp[i-1][j-1]
            else:
                dp[i][j] = min(dp[i-1][j], dp[i][j
                    -1], dp[i-1][j-1]) + 1
    return dp[m][n]
```

14 Unique Paths

14.1 Problem Statement

Given an m x n grid, find number of unique paths from top-left to bottom-right (right/down moves).

14.2 Dry Run on Test Cases

- **Test Case 1**: m = 3, n = 7 → Output: 28
- **Test Case 2**: m = 3, n = 2 → Output: 3
- **Test Case 3**: m = 1, n = 1 → Output: 1
- **Test Case 4**: m = 2, n = 2 → Output: 2

14.3 Algorithm (Memoization)

1. Recurse with memoization for paths to (i,j).
2. Base case: (0,0) returns 1, out of bounds returns 0.
3. Paths = sum of paths from above and left.

14.4 Algorithm (Tabulation)

1. dp[i][j] = paths to (i,j).
2. Initialize first row and column to 1.
3. dp[i][j] = dp[i-1][j] + dp[i][j-1].

Time Complexity: $O(m \cdot n)$ **Space Complexity**: $O(m \cdot n)$

14.5 Python Solution (Memoization)

```python
def unique_paths_memo(m, n):
    memo = {}
    def dp(i, j):
        if i < 0 or j < 0:
            return 0
        if i == 0 and j == 0:
            return 1
        if (i, j) in memo:
            return memo[(i, j)]
        memo[(i, j)] = dp(i-1, j) + dp(i, j-1)
        return memo[(i, j)]
    return dp(m-1, n-1)
```

14.6 Python Solution (Tabulation)

```python
def unique_paths(m, n):
    dp = [[1] * n for _ in range(m)]
    for i in range(1, m):
        for j in range(1, n):
            dp[i][j] = dp[i-1][j] + dp[i][j-1]
    return dp[m-1][n-1]
```

15 Unique Paths II

15.1 Problem Statement

Given an m x n grid with obstacles, find number of unique paths to bottom-right.

15.2 Dry Run on Test Cases

- **Test Case 1**: obstacleGrid = [[0,0,0],[0,1,0],[0,0,0]] → Output: 2
- **Test Case 2**: obstacleGrid = [[0,1],[0,0]] → Output: 1
- **Test Case 3**: obstacleGrid = [[1]] → Output: 0
- **Test Case 4**: obstacleGrid = [[0]] → Output: 1

15.3 Algorithm (Memoization)

1. Recurse with memoization for paths to (i,j).
2. Base case: obstacle or out of bounds returns 0, (0,0) returns 1 if no obstacle.
3. Paths = sum of paths from above and left.

15.4 Algorithm (Tabulation)

1. dp[i][j] = paths to (i,j).
2. Initialize dp[0][0] based on obstacle.
3. For each cell, dp[i][j] = dp[i-1][j] + dp[i][j-1] if no obstacle.

Time Complexity: $O(m \cdot n)$ **Space Complexity**: $O(m \cdot n)$

15.5 Python Solution (Memoization)

```python
def unique_paths_with_obstacles_memo(obstacleGrid):
    if not obstacleGrid or obstacleGrid[0][0] == 1:
        return 0
    m, n = len(obstacleGrid), len(obstacleGrid[0])
    memo = {}
    def dp(i, j):
        if i < 0 or j < 0 or obstacleGrid[i][j] ==
           1:
            return 0
        if i == 0 and j == 0:
            return 1
        if (i, j) in memo:
            return memo[(i, j)]
        memo[(i, j)] = dp(i-1, j) + dp(i, j-1)
        return memo[(i, j)]
    return dp(m-1, n-1)
```

15.6 Python Solution (Tabulation)

```python
def unique_paths_with_obstacles(obstacleGrid):
    if not obstacleGrid or obstacleGrid[0][0] == 1:
        return 0
    m, n = len(obstacleGrid), len(obstacleGrid[0])
    dp = [[0] * n for _ in range(m)]
    dp[0][0] = 1
    for i in range(m):
        for j in range(n):
            if obstacleGrid[i][j] == 1:
                continue
            if i > 0:
                dp[i][j] += dp[i-1][j]
            if j > 0:
                dp[i][j] += dp[i][j-1]
    return dp[m-1][n-1]
```

16 Minimum Path Sum

16.1 Problem Statement

Given an m x n grid of non-negative numbers, find minimum path sum from top-left to bottom-right.

16.2 Dry Run on Test Cases

- **Test Case 1**: grid = [[1,3,1],[1,5,1],[4,2,1]] → Output: 7 (1->3->1->1->1)
- **Test Case 2**: grid = [[1,2],[3,4]] → Output: 7
- **Test Case 3**: grid = [[1]] → Output: 1
- **Test Case 4**: grid = [] → Output: 0

16.3 Algorithm (Memoization)

1. Recurse with memoization for min sum to (i,j).
2. Base case: (0,0) returns grid[0][0], out of bounds returns inf.
3. Min sum = grid[i][j] + min(above, left).

16.4 Algorithm (Tabulation)

1. dp[i][j] = min sum to (i,j).
2. Initialize first row and column.
3. dp[i][j] = grid[i][j] + min(dp[i-1][j], dp[i][j-1]).

Time Complexity: $O(m \cdot n)$ **Space Complexity**: $O(m \cdot n)$

16.5 Python Solution (Memoization)

```python
def min_path_sum_memo(grid):
    if not grid:
        return 0
    m, n = len(grid), len(grid[0])
    memo = {}
    def dp(i, j):
        if i < 0 or j < 0:
            return float('inf')
        if i == 0 and j == 0:
            return grid[0][0]
        if (i, j) in memo:
            return memo[(i, j)]
        memo[(i, j)] = grid[i][j] + min(dp(i-1, j),
            dp(i, j-1))
        return memo[(i, j)]
    return dp(m-1, n-1)
```

16.6 Python Solution (Tabulation)

```python
def min_path_sum(grid):
    if not grid:
        return 0
    m, n = len(grid), len(grid[0])
    dp = [[0] * n for _ in range(m)]
    dp[0][0] = grid[0][0]
    for i in range(1, m):
        dp[i][0] = dp[i-1][0] + grid[i][0]
    for j in range(1, n):
        dp[0][j] = dp[0][j-1] + grid[0][j]
    for i in range(1, m):
        for j in range(1, n):
            dp[i][j] = grid[i][j] + min(dp[i-1][j],
                dp[i][j-1])
    return dp[m-1][n-1]
```

17 Decode Ways

17.1 Problem Statement

Given a string of digits, find number of ways to decode it into letters (1='A', 2='B', ..., 26='Z').

17.2 Dry Run on Test Cases

- **Test Case 1**: s = "12" → Output: 2 ("AB" or "L")
- **Test Case 2**: s = "226" → Output: 3 ("BBF", "VF", "BF")

- **Test Case 3**: s = "06" → Output: 0
- **Test Case 4**: s = "" → Output: 0

17.3 Algorithm (Memoization)

1. Recurse with memoization for ways from index i.
2. Base case: i=len(s) returns 1, invalid digit returns 0.
3. Add ways for single and double digit decodes.

17.4 Algorithm (Tabulation)

1. dp[i] = ways to decode prefix of length i.
2. Initialize dp[0]=1 if valid.
3. dp[i] += dp[i-1] if single digit valid, dp[i-2] if double digit valid.

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

17.5 Python Solution (Memoization)

```python
def num_decodings_memo(s):
    memo = {}
    def dp(i):
        if i == len(s):
            return 1
        if i in memo:
            return memo[i]
        if s[i] == '0':
            return 0
        result = dp(i + 1)
        if i + 1 < len(s) and (s[i] == '1' or (s[i]
            == '2' and s[i+1] <= '6')):
            result += dp(i + 2)
        memo[i] = result
        return result
    return 0 if not s or s[0] == '0' else dp(0)
```

17.6 Python Solution (Tabulation)

```python
def num_decodings(s):
    if not s or s[0] == '0':
        return 0
    dp = [0] * (len(s) + 1)
    dp[0] = 1
    dp[1] = 1
    for i in range(2, len(s) + 1):
        if s[i-1] != '0':
            dp[i] += dp[i-1]
        if s[i-2] == '1' or (s[i-2] == '2' and s[i-1] <= '6'):
            dp[i] += dp[i-2]
    return dp[len(s)]
```

18 Maximal Square

18.1 Problem Statement

Given a 2D binary matrix, find the size of the largest square of 1s.

18.2 Dry Run on Test Cases

- **Test Case 1**: matrix = [["1","0","1","0","0"],["1","0","1","1","1"],["1","1","1","1","1"],["1"
  $\rightarrow$ Output: 4
- **Test Case 2**: matrix = [["0","1"],["1","0"]] $\rightarrow$ Output: 1
- **Test Case 3**: matrix = [["0"]] $\rightarrow$ Output: 0
- **Test Case 4**: matrix = [] $\rightarrow$ Output: 0

18.3 Algorithm (Tabulation)

1. dp[i][j] = side length of max square ending at (i,j).
2. If matrix[i][j] = '1', dp[i][j] = min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]) + 1.
3. Track max side length.

Time Complexity: $O(m \cdot n)$ **Space Complexity**: $O(m \cdot n)$

18.4 Python Solution (Tabulation)

```python
def maximal_square(matrix):
    if not matrix or not matrix[0]:
        return 0
    m, n = len(matrix), len(matrix[0])
    dp = [[0] * (n + 1) for _ in range(m + 1)]
    max_side = 0
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if matrix[i-1][j-1] == '1':
                dp[i][j] = min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]) + 1
                max_side = max(max_side, dp[i][j])
    return max_side * max_side
```

19 Palindromic Substrings

19.1 Problem Statement

Given a string, count all palindromic substrings.

19.2 Dry Run on Test Cases

- **Test Case 1**: s = "abc" → Output: 3 ("a","b","c")
- **Test Case 2**: s = "aaa" → Output: 6 ("a","a","a","aa","aa","aaa")
- **Test Case 3**: s = "" → Output: 0
- **Test Case 4**: s = "a" → Output: 1

19.3 Algorithm (Tabulation)

1. dp[i][j] = True if substring i to j is palindrome.
2. Single characters are palindromes.
3. Check pairs, then longer substrings using dp[i+1][j-1].

Time Complexity: $O(n^2)$ **Space Complexity**: $O(n^2)$

19.4 Python Solution (Tabulation)

```python
def count_substrings(s):
    n = len(s)
    dp = [[False] * n for _ in range(n)]
    count = 0

    # Single characters
    for i in range(n):
        dp[i][i] = True
        count += 1

    # Pairs
    for i in range(n-1):
        if s[i] == s[i+1]:
            dp[i][i+1] = True
            count += 1

    # Longer substrings
    for length in range(3, n + 1):
        for i in range(n - length + 1):
            j = i + length - 1
            if s[i] == s[j] and dp[i+1][j-1]:
                dp[i][j] = True
                count += 1

    return count
```

20 Longest Palindromic Substring

20.1 Problem Statement

Given a string, find the longest palindromic substring.

20.2 Dry Run on Test Cases

- **Test Case 1**: s = "babad" → Output: "bab" or "aba"
- **Test Case 2**: s = "cbbd" → Output: "bb"
- **Test Case 3**: s = "" → Output: ""
- **Test Case 4**: s = "a" → Output: "a"

20.3 Algorithm (Tabulation)

1. dp[i][j] = True if substring i to j is palindrome.
2. Track longest palindrome with start index and length.
3. Check single, pairs, then longer substrings.

Time Complexity: $O(n^2)$ **Space Complexity**: $O(n^2)$

20.4 Python Solution (Tabulation)

```python
def longest_palindromic_substring(s):
    n = len(s)
    if n == 0:
        return ""
    dp = [[False] * n for _ in range(n)]
    start, max_len = 0, 1

    # Single characters
    for i in range(n):
        dp[i][i] = True

    # Pairs
    for i in range(n-1):
        if s[i] == s[i+1]:
            dp[i][i+1] = True
            start = i
            max_len = 2

    # Longer substrings
    for length in range(3, n + 1):
        for i in range(n - length + 1):
            j = i + length - 1
            if s[i] == s[j] and dp[i+1][j-1]:
                dp[i][j] = True
                if length > max_len:
                    start = i
                    max_len = length

    return s[start:start + max_len]
```

21 Number of 1 Bits

21.1 Problem Statement

Given an unsigned integer, return the number of 1 bits (Hamming weight).

21.2 Dry Run on Test Cases

- **Test Case 1**: n = 11 (1011) → Output: 3
- **Test Case 2**: n = 128 (10000000) → Output: 1
- **Test Case 3**: n = 0 → Output: 0
- **Test Case 4**: n = 4294967295 (111...1) → Output: 32

21.3 Algorithm

1. Use bitwise AND with 1 to check least significant bit.
2. Right shift number and count 1s.

Time Complexity: $O(1)$ (32 bits) **Space Complexity**: $O(1)$

21.4 Python Solution

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

22 Sum of Two Integers

22.1 Problem Statement

Given two integers, calculate their sum without using + or - operators.

22.2 Dry Run on Test Cases

- **Test Case 1**: a = 1, b = 2 → Output: 3
- **Test Case 2**: a = 2, b = 3 → Output: 5
- **Test Case 3**: a = 0, b = 0 → Output: 0
- **Test Case 4**: a = -2, b = 3 → Output: 1

22.3 Algorithm

1. Use XOR for sum without carry.
2. Use AND and left shift for carry.
3. Repeat until carry is 0, mask to 32 bits.

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

22.4 Python Solution

```python
def get_sum(a, b):
    mask = 0xffffffff
    while (b & mask) > 0:
        carry = ((a & b) << 1) & mask
        a = (a ^ b) & mask
        b = carry
    if (a >> 31) & 1:   # Handle negative numbers
        return ~(a ^ mask)
    return a
```

23 Single Number

23.1 Problem Statement

Given an array where every element appears twice except one, find the single number.

23.2 Dry Run on Test Cases

- **Test Case 1**: nums = [2,2,1] → Output: 1
- **Test Case 2**: nums = [4,1,2,1,2] → Output: 4
- **Test Case 3**: nums = [1] → Output: 1

- **Test Case 4**: nums = [] → Output: 0

23.3 Algorithm

1. Use XOR of all numbers.
2. Paired numbers cancel out, leaving single number.

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

23.4 Python Solution

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

24 Missing Number

24.1 Problem Statement

Given an array of [0,n] with one number missing, find the missing number.

24.2 Dry Run on Test Cases

- **Test Case 1**: nums = [3,0,1] → Output: 2
- **Test Case 2**: nums = [0,1] → Output: 2
- **Test Case 3**: nums = [9,6,4,2,3,5,7,0,1] → Output: 8
- **Test Case 4**: nums = [0] → Output: 1

24.3 Algorithm

1. Use XOR of all indices and numbers.
2. XOR with n to account for array length.
3. Result is the missing number.

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

24.4 Python Solution

```python
def missing_number(nums):
    result = len(nums)
    for i, num in enumerate(nums):
        result ^= i ^ num
    return result
```

25 Bitwise AND of Numbers Range

25.1 Problem Statement

Given a range [left, right], find the bitwise AND of all numbers.

25.2 Dry Run on Test Cases

- **Test Case 1**: left = 5, right = 7 → Output: 4 (5 6 7 = 4)
- **Test Case 2**: left = 0, right = 0 → Output: 0
- **Test Case 3**: left = 1, right = 2147483647 → Output: 0
- **Test Case 4**: left = 2, right = 3 → Output: 2

25.3 Algorithm

1. Shift left and right until they are equal.
2. Track shifts, shift result back.
3. Common prefix is the AND result.

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

25.4 Python Solution

```python
def range_bitwise_and(left, right):
    shift = 0
    while left != right:
        left >>= 1
        right >>= 1
        shift += 1
    return left << shift
```

26 Counting Bits

26.1 Problem Statement

Given an integer n, return an array of number of 1 bits for numbers 0 to n.

26.2 Dry Run on Test Cases

- **Test Case 1**: n = 2 → Output: [0,1,1]
- **Test Case 2**: n = 5 → Output: [0,1,1,2,1,2]
- **Test Case 3**: n = 0 → Output: [0]
- **Test Case 4**: n = 1 → Output: [0,1]

26.3 Algorithm

1. For each i, use i (i-1) to remove least significant 1.
2. dp[i] = dp[i (i-1)] + 1.

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

26.4 Python Solution

```python
def count_bits(n):
    dp = [0] * (n + 1)
    for i in range(1, n + 1):
        dp[i] = dp[i & (i-1)] + 1
    return dp
```

27 Reverse Bits

27.1 Problem Statement

Given a 32-bit unsigned integer, reverse its bits.

27.2 Dry Run on Test Cases

- **Test Case 1**: n = 43261596 (00000010100101000001111010011100) → Output: 964176192 (00111001011110000010100101000000)
- **Test Case 2**: n = 4294967295 (11111111111111111111111111111111) → Output: 4294967295
- **Test Case 3**: n = 0 → Output: 0
- **Test Case 4**: n = 1 → Output: 2147483648

27.3 Algorithm

1. Initialize result = 0.
2. For 32 bits, shift result left, add n's least significant bit, shift n right.

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

27.4 Python Solution

```python
def reverse_bits(n):
    result = 0
    for _ in range(32):
        result = (result << 1) | (n & 1)
        n >>= 1
    return result
```

28 Power of Two

28.1 Problem Statement

Given an integer, determine if it is a power of 2.

28.2 Dry Run on Test Cases

- **Test Case 1**: n = 1 → Output: True
- **Test Case 2**: n = 16 → Output: True
- **Test Case 3**: n = 3 → Output: False
- **Test Case 4**: n = 0 → Output: False

28.3 Algorithm

1. Check if n > 0 and n (n-1) == 0.
2. Powers of 2 have exactly one 1 bit.

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

28.4 Python Solution

```python
def is_power_of_two(n):
    return n > 0 and n & (n - 1) == 0
```

29 Power of Three

29.1 Problem Statement

Given an integer, determine if it is a power of 3.

29.2 Dry Run on Test Cases

- **Test Case 1**: n = 27 → Output: True
- **Test Case 2**: n = 0 → Output: False
- **Test Case 3**: n = 9 → Output: True
- **Test Case 4**: n = 45 → Output: False

29.3 Algorithm

1. Check if n > 0 and 3^19 **Time Complexity**: $O(1)$ **Space Complexity**: $O(1)$

29.4 Python Solution

```python
def is_power_of_three(n):
    return n > 0 and 1162261467 % n == 0  # 3^19 =
        1162261467
```

30 Power of Four

30.1 Problem Statement

Given an integer, determine if it is a power of 4.

30.2 Dry Run on Test Cases

- **Test Case 1**: n = 16 → Output: True
- **Test Case 2**: n = 5 → Output: False
- **Test Case 3**: n = 1 → Output: True
- **Test Case 4**: n = 0 → Output: False

30.3 Algorithm

1. Check if n > 0 and n (n-1) == 0 (power of 2).
2. Check if 1 bits are in odd positions (n 0x55555555 != 0).

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

30.4 Python Solution

```python
def is_power_of_four(n):
    return n > 0 and n & (n - 1) == 0 and n & 0
        x55555555 != 0
```