04. Solved Coding PYQs - Bitwise, Dynamic Programming & Matrix Problems

> **Module Focus**: High-frequency Bitwise, DP, and Matrix Manipulation Problems
> **Target**: 100% Test Case Pass Rate (Handling Integer Overflow, Negative Inputs, Edge Cases)

Problem 1: Find Two Non-Repeating Elements in Array (Bitwise XOR)

Problem Statement

Given an array where every element occurs twice except for **two numbers** which appear only once, find those two non-repeating numbers in $O(N)$ time complexity and $O(1)$ auxiliary space.

Solution Logic

1. XOR all elements in array. All repeating numbers cancel out ($x \oplus x = 0$). Result $X = A \oplus B$ where A, B are the two unique numbers.
2. Find the rightmost set bit in X using `mask = X & (-X)`.
3. Divide elements into two groups based on whether that bit is set or not.
4. XORing each group independently yields A and B.

Solution Implementations

C++ Solution

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

pair<int, int> findTwoUnique(const vector<int>& nums) {
    int xorSum = 0;
    for (int num : nums) {
        xorSum ^= num;
    }

    // Get rightmost set bit mask
    unsigned int rightmostBit = (unsigned int)xorSum & (-(unsigned int)xorSum);
```

```
    int num1 = 0, num2 = 0;
    for (int num : nums) {
        if ((unsigned int)num & rightmostBit) {
            num1 ^= num;
        } else {
            num2 ^= num;
        }
    }

    if (num1 > num2) swap(num1, num2);
    return {num1, num2};
}

int main() {
    vector<int> nums = {2, 4, 7, 9, 2, 4};
    pair<int, int> result = findTwoUnique(nums);
    cout << "Two Unique Numbers: " << result.first << " and " << result.second << endl;
    // Output: 7 and 9
    return 0;
}
```

Java Solution

```
import java.util.Arrays;

public class TwoUniqueElements {
    public static int[] findTwoUnique(int[] nums) {
        int xorSum = 0;
        for (int num : nums) {
            xorSum ^= num;
        }

        // Rightmost set bit mask
        int rightmostBit = xorSum & (-xorSum);

        int num1 = 0, num2 = 0;
        for (int num : nums) {
            if ((num & rightmostBit) != 0) {
                num1 ^= num;
            } else {
                num2 ^= num;
            }
        }

        int[] res = new int[]{num1, num2};
        Arrays.sort(res);
        return res;
    }
```

```
    public static void main(String[] args) {
        int[] nums = {2, 4, 7, 9, 2, 4};
        int[] result = findTwoUnique(nums);
        System.out.println("Two Unique Numbers: " + result[0] + " and " + result[1]);
        // Output: 7 and 9
    }
}
```

Python Solution

```
def find_two_unique(nums):
    xor_sum = 0
    for num in nums:
        xor_sum ^= num

    # Rightmost set bit mask
    rightmost_bit = xor_sum & (-xor_sum)

    num1, num2 = 0, 0
    for num in nums:
        if num & rightmost_bit:
            num1 ^= num
        else:
            num2 ^= num

    return sorted([num1, num2])

# Driver test
nums = [2, 4, 7, 9, 2, 4]
print("Two Unique Numbers:", find_two_unique(nums))  # Output: [7, 9]
```

Complexity Analysis

- **Time Complexity**: $\mathcal{O}(N)$ — Two passes over array of size N.
- **Space Complexity**: $\mathcal{O}(1)$ — In-place XOR bit manipulation.

Problem 2: Rotate Matrix 90 Degrees In-Place (Clockwise & Anti-Clockwise)

Problem Statement

Given an $N \times N$ 2D matrix representing an image, rotate the image by **90 degrees clockwise** and **90 degrees anti-clockwise** in-place without allocating another 2D matrix.

Solution Logic

1. Clockwise Rotation (90°)

1. **Transpose Matrix**: Swap `matrix[i][j]` with `matrix[j][i]` for $i < j$.
2. **Reverse Each Row**: Reverse elements of every row horizontally.

2. Anti-Clockwise (Counter-Clockwise) Rotation (90°)

1. **Transpose Matrix**: Swap `matrix[i][j]` with `matrix[j][i]` for $i < j$.
2. **Reverse Each Column**: Reverse elements of every column vertically (or reverse rows first, then transpose).

Solution Implementations

C++ Solution

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// Rotate 90 degrees Clockwise
void rotateClockwise(vector<vector<int>>& matrix) {
    int n = matrix.size();
    // Step 1: Transpose matrix
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            swap(matrix[i][j], matrix[j][i]);
        }
    }
    // Step 2: Reverse each row
    for (int i = 0; i < n; i++) {
        reverse(matrix[i].begin(), matrix[i].end());
    }
}

// Rotate 90 degrees Anti-Clockwise
void rotateAntiClockwise(vector<vector<int>>& matrix) {
    int n = matrix.size();
    // Step 1: Transpose matrix
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            swap(matrix[i][j], matrix[j][i]);
        }
    }
    // Step 2: Reverse each column
```

```
    for (int j = 0; j < n; j++) {
        for (int i = 0, k = n - 1; i < k; i++, k--) {
            swap(matrix[i][j], matrix[k][j]);
        }
    }
}

int main() {
    vector<vector<int>> matrix = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };

    vector<vector<int>> matCW = matrix;
    rotateClockwise(matCW);
    cout << "90 Deg Clockwise Rotated Matrix:\n";
    for (const auto& row : matCW) {
        for (int val : row) cout << val << " ";
        cout << "\n";
    }

    vector<vector<int>> matACW = matrix;
    rotateAntiClockwise(matACW);
    cout << "\n90 Deg Anti-Clockwise Rotated Matrix:\n";
    for (const auto& row : matACW) {
        for (int val : row) cout << val << " ";
        cout << "\n";
    }
    return 0;
}
```

Java Solution

```
import java.util.Arrays;

public class MatrixRotator {
    // 90 Degrees Clockwise
    public static void rotateClockwise(int[][] matrix) {
        int n = matrix.length;
        // Step 1: Transpose
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                int temp = matrix[i][j];
                matrix[i][j] = matrix[j][i];
                matrix[j][i] = temp;
            }
        }
        // Step 2: Reverse each row
```

```
        for (int i = 0; i < n; i++) {
            int left = 0, right = n - 1;
            while (left < right) {
                int temp = matrix[i][left];
                matrix[i][left] = matrix[i][right];
                matrix[i][right] = temp;
                left++;
                right--;
            }
        }
    }

    // 90 Degrees Anti-Clockwise
    public static void rotateAntiClockwise(int[][] matrix) {
        int n = matrix.length;
        // Step 1: Transpose
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                int temp = matrix[i][j];
                matrix[i][j] = matrix[j][i];
                matrix[j][i] = temp;
            }
        }
        // Step 2: Reverse each column
        for (int j = 0; j < n; j++) {
            int top = 0, bottom = n - 1;
            while (top < bottom) {
                int temp = matrix[top][j];
                matrix[top][j] = matrix[bottom][j];
                matrix[bottom][j] = temp;
                top++;
                bottom--;
            }
        }
    }

    public static void main(String[] args) {
        int[][] matrix = {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };

        int[][] cw = Arrays.stream(matrix).map(int[]::clone).toArray(int[][]::new);
        rotateClockwise(cw);
        System.out.println("90 Deg Clockwise Rotated Matrix:");
        for (int[] row : cw) System.out.println(Arrays.toString(row));

        int[][] acw = Arrays.stream(matrix).map(int[]::clone).toArray(int[][]::new);
        rotateAntiClockwise(acw);
```

```
        System.out.println("90 Deg Anti-Clockwise Rotated Matrix:");
        for (int[] row : acw) System.out.println(Arrays.toString(row));
    }
}
```

Python Solution

```
def rotate_clockwise(matrix):
    n = len(matrix)
    # Step 1: Transpose
    for i in range(n):
        for j in range(i + 1, n):
            matrix[i][j], matrix[j][i] = matrix[j][i], matrix[i][j]
    # Step 2: Reverse each row
    for i in range(n):
        matrix[i].reverse()

def rotate_anti_clockwise(matrix):
    n = len(matrix)
    # Step 1: Transpose
    for i in range(n):
        for j in range(i + 1, n):
            matrix[i][j], matrix[j][i] = matrix[j][i], matrix[i][j]
    # Step 2: Reverse each column
    for j in range(n):
        for i in range(n // 2):
            matrix[i][j], matrix[n - 1 - i][j] = matrix[n - 1 - i][j], matrix[i][j]

# Driver test
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

cw = [row[:] for row in matrix]
rotate_clockwise(cw)
print("90 Deg Clockwise Rotated Matrix:", cw)

acw = [row[:] for row in matrix]
rotate_anti_clockwise(acw)
print("90 Deg Anti-Clockwise Rotated Matrix:", acw)
```

Complexity Analysis

- **Time Complexity**: $\mathcal{O}(N^2)$ — Visiting every cell twice (transpose + row/column reversal).
- **Space Complexity**: $\mathcal{O}(1)$ — In-place rotation.

Problem 3: 0/1 Knapsack Problem (Dynamic Programming)

Problem Statement

Given weights `wt[]` and values `val[]` of N items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. Each item can either be picked or not picked (0/1 choice).

Solution Implementations

C++ Solution

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int knapsack(int W, const vector<int>& wt, const vector<int>& val, int n) {
    vector<int> dp(W + 1, 0);

    for (int i = 0; i < n; i++) {
        for (int w = W; w >= wt[i]; w--) {
            dp[w] = max(dp[w], val[i] + dp[w - wt[i]]);
        }
    }
    return dp[W];
}

int main() {
    vector<int> val = {60, 100, 120};
    vector<int> wt = {10, 20, 30};
    int W = 50;
    int n = val.size();
    cout << "Maximum Value: " << knapsack(W, wt, val, n) << endl; // Output: 220
    return 0;
}
```

Java Solution

```
public class Knapsack {
    public static int knapsack(int W, int[] wt, int[] val, int n) {
        int[] dp = new int[W + 1];

        for (int i = 0; i < n; i++) {
            for (int w = W; w >= wt[i]; w--) {
                dp[w] = Math.max(dp[w], val[i] + dp[w - wt[i]]);
            }
```

```
        }
        return dp[W];
    }

    public static void main(String[] args) {
        int[] val = {60, 100, 120};
        int[] wt = {10, 20, 30};
        int W = 50;
        int n = val.length;
        System.out.println("Maximum Value: " + knapsack(W, wt, val, n)); // Output: 220
    }
}
```

Python Solution

```
def knapsack(W, wt, val, n):
    dp = [0] * (W + 1)

    for i in range(n):
        for w in range(W, wt[i] - 1, -1):
            dp[w] = max(dp[w], val[i] + dp[w - wt[i]])

    return dp[W]

# Driver test
val = [60, 100, 120]
wt = [10, 20, 30]
W = 50
n = len(val)
print("Maximum Value:", knapsack(W, wt, val, n))  # Output: 220
```

Complexity Analysis

- **Time Complexity**: $\mathcal{O}(N \times W)$ — Nested loop over N items and W capacity.
- **Space Complexity**: $\mathcal{O}(W)$ — Space-optimized 1D DP table.

Problem 4: Count Set Bits in an Integer (Popcount)

Problem Statement

Write an efficient function to count the number of set bits (1s) in a given positive integer N.

Solution Implementations

C++ Solution

```
#include <iostream>

using namespace std;

int countSetBits(int n) {
    int count = 0;
    while (n > 0) {
        n &= (n - 1); // Clears the rightmost set bit (Brian Kernighan's Algorithm)
        count++;
    }
    return count;
}

int main() {
    int n = 29; // Binary: 11101
    cout << "Set Bits Count: " << countSetBits(n) << endl; // Output: 4
    return 0;
}
```

Java Solution

```
public class CountSetBits {
    public static int countSetBits(int n) {
        int count = 0;
        while (n > 0) {
            n &= (n - 1); // Clears the rightmost set bit
            count++;
        }
        return count;
    }

    public static void main(String[] args) {
        int n = 29; // Binary: 11101
        System.out.println("Set Bits Count: " + countSetBits(n)); // Output: 4
    }
}
```

Python Solution

```
def count_set_bits(n: int) -> int:
    count = 0
    while n > 0:
        n &= (n - 1)  # Clears the rightmost set bit
```

```
        count += 1
    return count

# Driver test
n = 29  # Binary: 11101
print("Set Bits Count:", count_set_bits(n))  # Output: 4
```

Complexity Analysis

- **Time Complexity**: $\mathcal{O}(K)$ — Where K is the number of set bits in N ($\le \log_2 N$).
- **Space Complexity**: $\mathcal{O}(1)$ — Uses zero auxiliary memory.

Problem 5: Array Transformation (Simulated Neighbor Modification)

Problem Statement

Given an array of integers `arr`, modify elements based on neighbor comparison: - An element `arr[i]` is **incremented by 1** if `arr[i-1] > arr[i]` and `arr[i] < arr[i+1]`. - An element `arr[i]` is **decremented by 1** if `arr[i-1] < arr[i]` and `arr[i] > arr[i+1]`. - The first (`arr[0]`) and last (`arr[n-1]`) elements never change. - Repeat transformations until no further elements change (array stabilizes). Return the stable array.

Solution Logic

1. Create a copy of `arr` at each iteration step to avoid using partially transformed values in the same step.
2. Iterate from index `1` to `n-2`:
3. If `arr[i] < arr[i-1]` and `arr[i] < arr[i+1]`, set `next_arr[i] = arr[i] + 1`.
4. Else if `arr[i] > arr[i-1]` and `arr[i] > arr[i+1]`, set `next_arr[i] = arr[i] - 1`.
5. If `next_arr` equals `arr`, the array has stabilized $\implies$ break loop and return `arr`.

Solution Implementations

C++ Solution

```
#include <iostream>
#include <vector>

using namespace std;

vector<int> transformArray(vector<int>& arr) {
    int n = arr.size();
    if (n <= 2) return arr;

    bool changed = true;
    while (changed) {
```

```
        changed = false;
        vector<int> nextArr = arr;
        for (int i = 1; i < n - 1; i++) {
            if (arr[i] < arr[i - 1] && arr[i] < arr[i + 1]) {
                nextArr[i]++;
                changed = true;
            } else if (arr[i] > arr[i - 1] && arr[i] > arr[i + 1]) {
                nextArr[i]--;
                changed = true;
            }
        }
        arr = nextArr;
    }
    return arr;
}

int main() {
    vector<int> arr = {6, 2, 3, 4, 1, 5};
    vector<int> result = transformArray(arr);
    cout << "Transformed Array: ";
    for (int x : result) cout << x << " ";
    cout << "\n";
    // Output: 6 3 3 3 3 5
    return 0;
}
```

Java Solution

```
import java.util.Arrays;

public class ArrayTransform {
    public static int[] transformArray(int[] arr) {
        int n = arr.length;
        if (n <= 2) return arr;

        boolean changed = true;
        while (changed) {
            changed = false;
            int[] nextArr = arr.clone();
            for (int i = 1; i < n - 1; i++) {
                if (arr[i] < arr[i - 1] && arr[i] < arr[i + 1]) {
                    nextArr[i]++;
                    changed = true;
                } else if (arr[i] > arr[i - 1] && arr[i] > arr[i + 1]) {
                    nextArr[i]--;
                    changed = true;
                }
            }
            arr = nextArr;
```

```
        }
        return arr;
    }

    public static void main(String[] args) {
        int[] arr = {6, 2, 3, 4, 1, 5};
        int[] result = transformArray(arr);
        System.out.println("Transformed Array: " + Arrays.toString(result));
        // Output: [6, 3, 3, 3, 3, 5]
    }
}
```

Python Solution

```
def transform_array(arr: list[int]) -> list[int]:
    n = len(arr)
    if n <= 2:
        return arr

    changed = True
    while changed:
        changed = False
        next_arr = list(arr)
        for i in range(1, n - 1):
            if arr[i] < arr[i - 1] and arr[i] < arr[i + 1]:
                next_arr[i] += 1
                changed = True
            elif arr[i] > arr[i - 1] and arr[i] > arr[i + 1]:
                next_arr[i] -= 1
                changed = True
        arr = next_arr
    return arr

# Driver test
arr = [6, 2, 3, 4, 1, 5]
print("Transformed Array:", transform_array(arr))  # Output: [6, 3, 3, 3, 3, 5]
```

Complexity Analysis

- **Time Complexity**: $\mathcal{O}(D \times N)$ — Where D is the number of transformation cycles until stabilization ($D \le \max(arr)$).
- **Space Complexity**: $\mathcal{O}(N)$ — Auxiliary copy array to avoid in-step state corruption.

Problem 6: Run-Length String Compression

Problem Statement

Implement a method to perform basic string compression using the counts of repeated characters (e.g., `"aabccccccaaa"` $\to$ `"a2b1c5a3"`). If the "compressed" string would not become smaller than the original string, your method should return the original string. You can assume the string has only uppercase and lowercase letters (`a-z`, `A-Z`).

Solution Logic

1. Handle null or empty string edge cases immediately.
2. Use a dynamic string buffer (`StringBuilder` in Java, `std::string` in C++, list join in Python).
3. Loop through characters, keeping track of consecutive frequency `count`.
4. Append character and count whenever character changes or string end is reached.
5. Compare length of compressed string vs original string and return the shorter one.

Solution Implementations

C++ Solution

```
#include <iostream>
#include <string>

using namespace std;

string compressString(const string& s) {
    if (s.empty()) return s;

    string compressed = "";
    int count = 1;

    for (size_t i = 0; i < s.length(); i++) {
        if (i + 1 < s.length() && s[i] == s[i + 1]) {
            count++;
        } else {
            compressed += s[i];
            compressed += to_string(count);
            count = 1;
        }
    }

    return (compressed.length() < s.length()) ? compressed : s;
}

int main() {
```

```
    string str1 = "aabcccccaaa";
    string str2 = "abcd";
    cout << str1 << " -> " << compressString(str1) << "\n"; // Output: a2b1c5a3
    cout << str2 << " -> " << compressString(str2) << "\n"; // Output: abcd (original)
    return 0;
}
```

Java Solution

```
public class StringCompressor {
    public static String compressString(String s) {
        if (s == null || s.isEmpty()) return s;

        StringBuilder compressed = new StringBuilder();
        int count = 1;

        for (int i = 0; i < s.length(); i++) {
            if (i + 1 < s.length() && s.charAt(i) == s.charAt(i + 1)) {
                count++;
            } else {
                compressed.append(s.charAt(i));
                compressed.append(count);
                count = 1;
            }
        }

        String result = compressed.toString();
        return (result.length() < s.length()) ? result : s;
    }

    public static void main(String[] args) {
        String str1 = "aabcccccaaa";
        String str2 = "abcd";
        System.out.println(str1 + " -> " + compressString(str1)); // Output: a2b1c5a3
        System.out.println(str2 + " -> " + compressString(str2)); // Output: abcd
    }
}
```

Python Solution

```
def compress_string(s: str) -> str:
    if not s:
        return s

    compressed = []
    count = 1

    for i in range(len(s)):
```

```
        if i + 1 < len(s) and s[i] == s[i + 1]:
            count += 1
        else:
            compressed.append(s[i] + str(count))
            count = 1

    result = "".join(compressed)
    return result if len(result) < len(s) else s

# Driver test
str1 = "aabccccccaaa"
str2 = "abcd"
print(f"{str1} -> {compress_string(str1)}")  # Output: a2b1c5a3
print(f"{str2} -> {compress_string(str2)}")  # Output: abcd
```

Complexity Analysis

- **Time Complexity**: $\mathcal{O}(N)$ — Single pass over string of length N.
- **Space Complexity**: $\mathcal{O}(N)$ — Space for storing compressed string output.