Documentation for Leetcode 474: Ones and Zeroes

Table of Contents

1. **Problem Statement**

   You're given:

   - An array of binary strings strs
   - Two integers m and n

   Your task is to find the size of the largest subset of strs such that the total number of '0's is at most m and the total number of '1's is at most n.

   Constraints:

   - $1 \leq$ strs.length ≤ 600
   - $1 \leq$ strs[i].length ≤ 100
   - strs[i] contains only '0' and '1'
   - $1 \leq m, n \leq 100$

2. Intuition

This is a variant of the 0/1 Knapsack problem, where:

- Each string has a "cost" in terms of the number of 0s and 1s.
- We are bounded by two constraints: total 0s ≤ m, and total 1s ≤ n.

We aim to maximize the number of strings selected within these constraints.

3. Key Observations

- The count of 0s and 1s in each string acts like a weight.
- We need to track the maximum number of items (strings) that can be picked without exceeding these weights.
- The problem is not about summing values, but about maximizing the count of included elements.

4. Approach

- Use Dynamic Programming (DP).
- Define a 2D DP array dp[m+1][n+1] where dp[i][j] = max number of strings that can be picked with i 0s and j 1s.
- For each string:
  - Count its zero and one.
  - Update the dp table in reverse order to avoid counting the same string multiple times.
- Finally, return dp[m][n].

5. Edge Cases

- Empty strs list: Return 0.
- Strings with only 0s or only 1s.
- m = 0 or n = 0: Only strings with no 0s or 1s can be included.
- Strings with counts greater than m or n must be excluded.

6. Complexity Analysis

Time Complexity:

- O(L * m * n)
  - L = length of strs
  - Each string updates up to m * n entries in the DP table

Space Complexity:

- O(m * n)
  - We use a 2D DP table of size (m+1) x (n+1)

7. Alternative Approaches

- Brute-force recursion with memoization:
  - Explore all subsets with recursion and memoize results.
  - More complex to manage and less efficient for large inputs.
- 3D DP: Use dp[i][j][k] for tracking up to i-th string. But this is unnecessary; 2D DP is optimal.

8. Test Cases

✅ Test Case 1:

Input: strs = ["10","0001","111001","1","0"], m = 5, n = 3
Output: 4
Explanation: Subset = ["10","0001","1","0"]

✅ Test Case 2:

Input: strs = ["10","0","1"], m = 1, n = 1
Output: 2
Explanation: Subset = ["0", "1"]

✅ Test Case 3:

Input: strs = ["0","0","1","1"], m = 2, n = 2

Output: 4

✅ Test Case 4:

Input: strs = ["10"], m = 0, n = 1

Output: 0

9. Final Thoughts

- This problem teaches how to extend classical knapsack DP to multiple constraints.
- Always update the DP table in reverse when doing 0/1 knapsack to avoid reusing items.
- Efficient even for large inputs due to optimal O(m * n * L) complexity.