474. Ones and Zeroes

String]

Dynamic Programming

Problem Description

Array

Medium

string consists only of '0's and '1's. We also have two integer constraints, m and n, which are the maximum number of '0's and '1's allowed in our subset, respectively. The key part to understand here is:

• A subset means every element in our selected group of strings must also exist in the original group (strs), but not necessarily all elements in

In this problem from LeetCode, we are tasked with finding the largest subset of binary strings from a given array strs. A binary

strs have to be in our subset.

- When creating the largest subset, the total count of '0's in all the strings of our subset cannot exceed m, and the total count of '1's cannot exceed n.
- Our goal is to identify the size of this largest subset. Size means the number of strings within it, not the length of each string.

We start by understanding that the choice of including each string can affect our ability to include other strings. If we include

While updating, we consider two cases for each cell f[i][j]:

(and thus having 0 '0's and 1's) has a size of 0`.

To solve this problem, we use a <u>dynamic programming</u> approach.

Generally, in dynamic programming, we try to solve smaller subproblems and use their results to construct solutions for larger subproblems. This concept is called optimal substructure.

a string with many '0's and '1's, it may prevent us from adding additional strings later. Thus, we have to make careful choices.

Here, we define a two-dimensional array f where f[i][j] represents the size of the largest subset we can form with at most

- i '0's and j '1's. The values of i range from 0 to m, and the values of j range from 0 to n, representing all possible constraints we might encounter.
- We initialize our array with zeros, as the largest subset with zero '0's and '1's is an empty subset, hence zero size. We iterate through each string in strs and count its '0's and '1's. Then, for each string, we update our array f in a decreasing manner, starting from m to the count of '0's in the current string (a) and from n to the count of '1's (b).
- Not including the current string, which means the f[i][j] would remain unchanged. Including the current string, where we have to look at the value in the cell that represents the leftover capacity (f[i - a][j - b]) after
- including this string and add 1 to that value to represent the current string being counted. We choose the maximum of these two choices at every step, which ensures that we always have the largest possible subset

After iterating through all strings and updating the array, the value of f[m][n] will give us the size of the largest subset

- for a given i and j.
- This <u>dynamic programming</u> solution is efficient as it avoids recalculating the largest subset sizes for every combination of '0's and '1's by building upon previously computed values.
- The implementation of the solution follows the <u>dynamic programming</u> approach to methodically work towards the final answer. Here's an in-depth walk-through of the pattern and the algorithm used:

lists. Each cell f[i][j] in this array represents the size of the largest subset with i '0's and j '1's. The + 1 in both dimensions is used to include the case where 0 '0's or '1's are used.

Data Structure: A two-dimensional list f is created with dimensions $(m + 1) \times (n + 1)$. In Python, this is realized as a list of

Initialization: The two-dimensional list f is initialized with zeroes, since the largest subset without considering any strings

Counting '0's and '1's: For each string s in strs, s.count("0") and s.count("1") are called to count the number of '0's (a) and

in our subset.

Example Walkthrough

filled with zeros, like so:

'0's and '1's can still be included) at each step.

Solution Approach

conforming to our constraints.

'1's (b) respectively. **Updating the DP Table**: We iterate over the list in reverse for i from m to a - 1 and j from n to b - 1. We do this because we

want to make sure that when we account for a new string, we are not overwriting cells that could affect the calculation of

cells later in the iteration. This is a common technique in dynamic programming known as avoiding "state contamination."

Choice: At each cell f[i][j], we attempt to include the current string. To do this, we compare the existing value f[i][j] (not including the current string) with f[i - a][j - b] + 1 (including the string). f[i - a][j - b] represents the largest subset possible with the remaining capacity after including the current string. We add 1 because we are including the current string

Taking the Maximum: We use Python's max function to always store the maximum of the two values. Thus, f[i][j] will

Result: After completely filling the two-dimensional list, f[m] [n] will give us the maximum size of our desired subset since it

always hold the size of the largest subset for the specific capacity represented by i and j.

m = 5 and n = 3. This means we cannot have more than 5 '0's and 3 '1's in our subset.

Counting and updating: We go through each string in strs and update our DP table.

represents the size of the largest subset under the full capacity of m '0's and n '1's. This implementation successfully leverages the central ideas of dynamic programming, namely optimal substructure (solving

bigger problems by relying on the solutions to smaller problems) and overlapping subproblems (saving computation by storing

intermediate results). The use of a two-dimensional DP table is crucial, as it allows tracking the state of the problem (how many

Let's take a small example to illustrate the solution approach. Assume we have the following array of binary strings strs: ["10", "0001", "111001", "1", "0"], and our integer constraints are

f = [[0, 0, 0, 0], [0, 0, 0, 0],[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]

∘ For the string "10", we have a=1 (the number of '0's) and b=1 (the number of '1's). We update cells in the range of i = 5 to 1 and j = 3 to

1. Initialization: We create a two-dimensional list f with dimensions (m + 1) x (n + 1), so f would be a 6×4 matrix, as we include zero counts. It's

1, comparing the existing value f[i][j] with f[i-1][j-1]+1 (since a=1 and b=1). After this string, our DP table update looks like this:

[0, 1, 1, 1]

[0, 0, 0, 0],

[0, 1, 1, 1],

[1, 1, 2, 2],

[1, 2, 2, 2],

[1, 2, 3, 3],

[1, 2, 3, 4]

Python

Solution Implementation

from typing import List

public class Solution {

for s in strings:

class Solution:

f = [

f = [

[0, 1, 1, 1], [0, 1, 1, 1], [0, 1, 1, 1],

[0, 1, 1, 1], // '10' considered for i=1 (up to 1 '0's used)

[0, 0, 0, 0], // No strings considered yet

• Applying a similar process for other strings: "0001", "111001", "1", and "0", updating f for the '0's and '1's in each and choosing the max value for each cell.

3. Final DP Table: Once we process all strings, our DP table will display the maximum number of strings that can be included for a given i number

of '0's and j number of '1's. For our case, we getf[m][n] as the result. Let's assume our final table after all updates looks like this:

'0's and '1's allowed. The subset, in this case, could be ["10", "0001", "1", "0"], which includes 4 strings, adheres to the

4. Result: From the last entry f[5][3], we see that the maximum size of the subset we can get under the given constraints is 4. Thus, with m = 5 and n = 3, we are able to include four strings from the array strs in our subset without exceeding the number of

constraints (`5 '0's and 3 '1's), and is the largest possible subset for these constraints.

def findMaxForm(self, strings: List[str], max_zeros: int, max_ones: int) -> int:

Count the number of zeros and ones in the current string

 $dp = [[0] * (max_ones + 1) for _ in range(max_zeros + 1)]$

zero_count, one_count = s.count("0"), s.count("1")

for zeros in range(max_zeros, zero_count - 1, -1):

for ones in range(max_ones, one_count - 1, -1):

// Count the number of zeros and ones in the current string

Iterate through each string in the input list

public int findMaxForm(String[] strs, int m, int n) {

// Iterate through each string in the input list

int[] count = countZerosAndOnes(s);

for (int i = m; i >= count[0]; --i) {

for (int j = n; j >= count[1]; ---j) {

// Helper function to count the number of zeros and ones in a string

// Return the maximum number of strings that can be formed

// Helper function to count the number of zeroes and ones in a string

// Since the total length minus zeroes gives the number of ones

return {countZeroes, static_cast<int>(str.size()) - countZeroes};

// First of the pair is number of zeroes, second is the number of ones

function findMaxForm(strings: string[], zeroLimit: number, oneLimit: number): number {

// Initialize a memoization table with dimensions (zeroLimit + 1) \times (oneLimit + 1).

// This table will help us keep track of the maximum number of strings we can include

// with given 'm' zeroes and 'n' ones

pair<int, int> countZeroesAndOnes(string& str) {

// given a specific limit of zeroes and ones.

const dpTable = Array.from({ length: zeroLimit + 1 }, () =>

Array.from({ length: oneLimit + 1 }, () => 0)

return dp[m][n];

private:

};

TypeScript

);

int[][] dp = new int[m + 1][n + 1];

private int[] countZerosAndOnes(String s) {

for (int i = 0; i < s.length(); ++i) {</pre>

++count[s.charAt(i) - '0'];

// Iterate through the characters of the string

int[] count = new int[2];

for (String s : strs) {

Initialize the DP table with dimensions (max_zeros + 1) by (max_ones + 1)

Update the DP table value for the current subproblem

Iterate over the DP table in reverse to avoid using a result before it's updated

// The main function to find maximum number of strings that can be formed with m zeros and n ones

// Loop over the dp array from bottom up considering the current string's zeros and ones

dp[i][j] = Math.max(dp[i][j], dp[i - count[0]][j - count[1]] + 1);

// Increment the respective count (0 or 1) based on the current character

dp[zeros][ones] = max(dp[zeros][ones], dp[zeros - zero_count][ones - one_count] + 1) # The answer is the value corresponding to using maximum zeros and ones return dp[max_zeros][max_ones] Java

// Initialize a DP table where f[i][j] will represent the max number of strings that can be formed with i zeros and j one

// Update the dp value with the higher value between the current and the new computed one

// Initialize a count array where the first element is the number of zeros and the second is the number of ones

```
// Return the result from the DP table which is the maximum number of strings that can be formed
return dp[m][n];
```

```
// Return the count array
        return count;
C++
#include <vector>
#include <string>
#include <algorithm>
#include <cstring> // For memset
using namespace std;
class Solution {
public:
    int findMaxForm(vector<string>& strs, int m, int n) {
       // Create a 2D array (dp) with dimensions m+1 and n+1
       // Initialize all elements to zero
       int dp[m + 1][n + 1];
       memset(dp, 0, sizeof(dp));
       // Iterate over each string in the given vector 'strs'
        for (auto& str : strs) {
            // Count the number of zeroes and ones in the current string
            pair<int, int> zeroOneCount = countZeroesAndOnes(str);
            int zeroes = zeroOneCount.first;
            int ones = zeroOneCount.second;
            // Iterate over the matrix in reverse, to avoid over-counting
            // when using previously computed sub-solutions
            for (int i = m; i >= zeroes; --i) {
                for (int j = n; j >= ones; --j) {
                    // Update the dp matrix by taking the maximum between:
                    // 1. Current cell value (previous computed max)
                    // 2. Value computed by including the current string
                    // Add 1 to the subproblem solution because
                    // we are including one more string
                    dp[i][j] = max(dp[i][j], dp[i - zeroes][j - ones] + 1);
```

```
// A helper function to count the number of zeroes and ones in a string.
// It returns a tuple [zeroCount, oneCount].
const countZeroesAndOnes = (str: string): [number, number] => {
    let zeroCount = 0;
    for (const char of str) {
        if (char === '0') {
            zeroCount++;
   return [zeroCount, str.length - zeroCount];
};
// Iterate through each string in the input array.
for (const str of strings) {
   // Count the number of zeroes and ones in the current string.
   const [zeroes, ones] = countZeroesAndOnes(str);
   // Update the dpTable in reverse to avoid overwriting data we still need to use.
    for (let i = zeroLimit; i >= zeroes; --i) {
        for (let j = oneLimit; j >= ones; --j) {
            // The maximum number of strings that can be included is either the current count
```

// that can be included with the remaining zeroes and ones.

int countZeroes = count_if(str.begin(), str.end(), [](char c) { return c == '0'; });

```
// The final result is stored in dpTable[zeroLimit][oneLimit], reflecting the maximum number
      // of strings we can include given the original zeroLimit and oneLimit.
      return dpTable[zeroLimit][oneLimit];
from typing import List
class Solution:
   def findMaxForm(self, strings: List[str], max_zeros: int, max_ones: int) -> int:
       # Initialize the DP table with dimensions (max_zeros + 1) by (max_ones + 1)
       dp = [[0] * (max_ones + 1) for _ in range(max_zeros + 1)]
       # Iterate through each string in the input list
        for s in strings:
           # Count the number of zeros and ones in the current string
            zero_count, one_count = s.count("0"), s.count("1")
```

Iterate over the DP table in reverse to avoid using a result before it's updated

dp[zeros][ones] = max(dp[zeros][ones], dp[zeros - zero_count][ones - one_count] + 1)

// or the count obtained by including the current string plus the count of strings

dpTable[i][j] = Math.max(dpTable[i][j], dpTable[i - zeroes][j - ones] + 1);

Time and Space Complexity Time Complexity:

Update the DP table value for the current subproblem

The answer is the value corresponding to using maximum zeros and ones

for zeros in range(max_zeros, zero_count - 1, -1):

return dp[max_zeros][max_ones]

for ones in range(max_ones, one_count - 1, -1):

```
over all strings in strs, and for each string, we iterate through a 2D array of size m * n in a nested loop fashion.
```

Space Complexity: The space complexity of the solution is 0(m * n), as we are constructing a 2D array f with m + 1 rows and n + 1 columns to store the intermediate results for dynamic programming. No other data structures are used that grow with the size of the input, so the space complexity is directly proportional to the size of the 2D array.

The time complexity of the given solution is 0(k * m * n), where k is the length of the input list strs, m is the maximum number of

zeroes, and n is the maximum number of ones that our subsets from strs can contain. This complexity arises because we iterate