<https://leetcode.com/problems/ones-and-zeroes/description/>

You are given an array of binary strings strs and two integers m and n.

Return *the size of the largest subset of strs such that there are* ***at most***m0*'s and* n1*'s in the subset*.

A set x is a **subset** of a set y if all elements of x are also elements of y.

**Example 1:**

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

Output: 4

Explanation: The largest subset with at most 5 0's and 3 1's is {"10", "0001", "1", "0"}, so the answer is 4.

Other valid but smaller subsets include {"0001", "1"} and {"10", "1", "0"}.

{"111001"} is an invalid subset because it contains 4 1's, greater than the maximum of 3.

**Example 2:**

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

Output: 2

Explanation: The largest subset is {"0", "1"}, so the answer is 2.

**Constraints:**

* 1 <= strs.length <= 600
* 1 <= strs[i].length <= 100
* strs[i] consists only of digits '0' and '1'.
* 1 <= m, n <= 100

**Attempt 1: 2023-11-11**

**Wrong Solution: The "Pick" call should not happen at all when 'm < 0 || n < 0'**

**The 'm < 0 || n < 0' here not stop the call on "pick" logic when 'm < 0 || n < 0' actually happened, the "pick" call anyway triggered and return 0 + 1, the additional 1 is fatal, because the "pick" call suppose not even should happen when 'm < 0 || n < 0' and only should return 0, to solve this, we need a working block condition**

Test out by:

strs = ["10","0001","111001","1","0"], m = 4, n = 3

Expected: 3

Output: 4

=====================================================================

class Solution {

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

return helper(strs, m, n, 0);

}

private int helper(String[] strs, int m, int n, int index) {

// The 'm < 0 || n < 0' here not stop the call on "pick" logic

// when 'm < 0 || n < 0' actually happened, the "pick" call

// anyway triggered and return 0 + 1, the additional 1 is

// fatal, because the "pick" call suppose not even should

// happen when 'm < 0 || n < 0' and only should return 0,

// to solve this, we need a working block condition

if(index >= strs.length || m < 0 || n < 0) {

return 0;

}

int index\_str\_ones = countOnes(strs[index]);

int index\_str\_zeros = strs[index].length() - index\_str\_ones;

// Pick element at index

// Don't forget to add 1 on subset size for current pick

int pick = helper(strs, m - index\_str\_zeros, n - index\_str\_ones, index + 1) + 1;

// Not pick element at index

int not\_pick = helper(strs, m, n, index + 1);

return Math.max(pick, not\_pick);

}

private int countOnes(String str) {

int count = 0;

for(char c : str.toCharArray()) {

if(c == '1') {

count++;

}

}

return count;

}

}

**Solution 1:  Native DFS (120 min, TLE 22/72)**

class Solution {

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

return helper(strs, m, n, 0);

}

private int helper(String[] strs, int m, int n, int index) {

// Only add condition as 'm < 0 || n < 0' here won't work and not stop

// the call when current element ones and zeros exceed m and n limitation

// instead we have to add below block condition separately, the consequence

// of not adding below block condition is the call will continue hit "pick"

// logic either m < index\_str\_zeros or n < index\_str\_ones, which will surely

// return 0 + 1 = 1 at bottom of recursion, the additional 1 is fatal since

// we don't even suppose to continue call "pick" logic in this scenario, the

// return should be 0, not 1

// Pick logic:

// helper(strs, m - index\_str\_zeros, n - index\_str\_ones, index + 1) + 1;

// Required block condition:

// if(m - index\_str\_zeros < 0 || n - index\_str\_ones < 0) {

// return helper(strs, m, n, index + 1);

//}

//if(index >= strs.length || m < 0 || n < 0) {

if(index >= strs.length) {

return 0;

}

int index\_str\_ones = countOnes(strs[index]);

int index\_str\_zeros = strs[index].length() - index\_str\_ones;

// If remain m or n less than current element's ones or zeros we have no choice,

// not able to pick it up and have to go ahead directly to next element if exist

if(m - index\_str\_zeros < 0 || n - index\_str\_ones < 0) {

return helper(strs, m, n, index + 1);

}

// Only when current element ones and zeros not exceed m and n limitation

// we have two choices: pick or not pick current element

// Pick element at index

// Don't forget to add 1 on subset size for current pick

int pick = helper(strs, m - index\_str\_zeros, n - index\_str\_ones, index + 1) + 1;

// Not pick element at index

int not\_pick = helper(strs, m, n, index + 1);

return Math.max(pick, not\_pick);

}

private int countOnes(String str) {

int count = 0;

for(char c : str.toCharArray()) {

if(c == '1') {

count++;

}

}

return count;

}

}

Time Complexity: O(2^N)

Space Complexity: O(2^N)

**Step by Step process:**

Initial status

------------------------------------------

strs = ["10", "0001", "111001", "1", "0"]

m = 4

n = 3

helper(strs, 4, 3, 0)

------------------------------------------

index = 0

"10" -> index\_str\_zeros = 1, index\_str\_ones = 1

m - 1 = 4 - 1 = 3 >= 0

n - 1 = 3 - 1 = 2 >= 0

have choice to pick / not pick

pick: helper(strs, 3, 2, 1) + 1

------------------------------------------

index = 1

"0001" -> index\_str\_zeros = 3, index\_str\_ones = 1

m - 1 = 3 - 3 = 0 >= 0

n - 1 = 2 - 1 = 1 >= 0

have choice to pick / not pick

pick: helper(strs, 0, 1, 2) + 1

------------------------------------------

index = 2

"111001" -> index\_str\_zeros = 2, index\_str\_ones = 4

m - 1 = 0 - 2 = -2 < 0

n - 1 = 1 - 4 = -3 < 0

no choice have to skip

return helper(strs, 0, 1, 3)

------------------------------------------

index = 3

"1" -> index\_str\_zeros = 0, index\_str\_ones = 1

m - 0 = 0 - 0 = 0 >= 0

n - 1 = 1 - 1 = 0 >= 0

have choice to pick / not pick

pick: helper(strs, 0, 0, 4) + 1

------------------------------------------

index = 4

"0" -> index\_str\_zeros = 1, index\_str\_ones = 0

m - 1 = 0 - 1 = -1 < 0

n - 0 = 0 - 0 = 0 >= 0

no choice have to skip

return helper(strs, 0, 0, 5)

------------------------------------------

index = 5

return 0

------------------------------------------

index = 4

return helper(strs, 0, 0, 5) = 0

------------------------------------------

index = 3

pick = helper(strs, 0, 0, 4) + 1 = 1

not pick: helper(strs, 0, 1, 4)

------------------------------------------

index = 4

"0" -> index\_str\_zeros = 1, index\_str\_ones = 0

m - 1 = 0 - 1 = -1 < 0

n - 0 = 1 - 0 = 1 >= 0

no choice have to skip

return helper(strs, 0, 1, 5)

------------------------------------------

index = 5

return 0

------------------------------------------

index = 4

return helper(strs, 0, 1, 5) = 0

------------------------------------------

index = 3

pick = 1

not pick = helper(strs, 0, 1, 4) = 0

return Math.max(1, 0) = 1

------------------------------------------

index = 2

return helper(strs, 0, 1, 3) = 1

------------------------------------------

index = 1

pick = helper(strs, 0, 1, 2) + 1 = 2

not pick: helper(strs, 3, 2, 2)

------------------------------------------

index = 2

"111001" -> index\_str\_zeros = 2, index\_str\_ones = 4

m - 1 = 3 - 2 = 1 >= 0

n - 1 = 2 - 4 = -2 < 0

no choice have to skip

return helper(strs, 3, 2, 3)

------------------------------------------

index = 3

"1" -> index\_str\_zeros = 0, index\_str\_ones = 1

m - 0 = 3 - 0 = 3 >= 0

n - 1 = 2 - 1 = 1 >= 0

have choice to pick / not pick

pick: helper(strs, 3, 1, 4) + 1

------------------------------------------

index = 4

"0" -> index\_str\_zeros = 1, index\_str\_ones = 0

m - 1 = 3 - 1 = 2 >= 0

n - 0 = 1 - 0 = 1 >= 0

have choice to pick / not pick

pick: helper(strs, 2, 1, 5) + 1

------------------------------------------

index = 5

return 0

------------------------------------------

index = 4

pick = helper(strs, 2, 1, 5) + 1 = 1

not pick: helper(strs, 3, 1, 5)

------------------------------------------

index = 5

return 0

------------------------------------------

index = 4

pick = 1

not pick = helper(strs, 3, 1, 5) = 0

return Math.max(1, 0) = 1

------------------------------------------

index = 3

pick = helper(strs, 3, 1, 4) + 1 = 2

not pick: helper(strs, 3, 2, 4)

------------------------------------------

index = 4

"0" -> index\_str\_zeros = 1, index\_str\_ones = 0

m - 1 = 3 - 1 = 2 >= 0

n - 0 = 2 - 0 = 2 >= 0

have choice to pick / not pick

pick: helper(strs, 2, 2, 5) + 1

------------------------------------------

index = 5

return 0

------------------------------------------

index = 4

pick = helper(strs, 2, 2, 5) + 1 = 1

not pick: helper(strs, 3, 2, 5)

------------------------------------------

index = 5

return 0

------------------------------------------

index = 4

pick = 1

not pick = helper(strs, 3, 2, 5) = 0

return Math.max(1, 0) = 1

------------------------------------------

index = 3

pick = 2

not pick = helper(strs, 3, 2, 4) = 1

return Math.max(2, 1) = 2

------------------------------------------

index = 2

return helper(strs, 3, 2, 3) = 2

------------------------------------------

index = 1

pick = 2

not pick = helper(strs, 3, 2, 2) = 2

return Math.max(2, 2) = 2

------------------------------------------

index = 0

pick = helper(strs, 3, 2, 1) + 1 = 3

not pick: helper(strs, 4, 3, 1)

------------------------------------------

index = 1

"0001" -> index\_str\_zeros = 3, index\_str\_ones = 1

m - 1 = 4 - 3 = 1 >= 0

n - 1 = 3 - 1 = 2 >= 0

have choice to pick / not pick

pick: helper(strs, 1, 2, 2) + 1

------------------------------------------

index = 2

"111001" -> index\_str\_zeros = 2, index\_str\_ones = 4

m - 1 = 1 - 2 = -1 < 0

n - 1 = 2 - 4 = -2 < 0

no choice have to skip

return helper(strs, 1, 2, 3)

------------------------------------------

index = 3

"1" -> index\_str\_zeros = 0, index\_str\_ones = 1

m - 0 = 1 - 0 = 1 >= 0

n - 1 = 2 - 1 = 1 >= 0

have choice to pick / not pick

pick: helper(strs, 1, 1, 4) + 1

------------------------------------------

index = 4

"0" -> index\_str\_zeros = 1, index\_str\_ones = 0

m - 1 = 1 - 1 = 0 >= 0

n - 0 = 1 - 0 = 1 >= 0

have choice to pick / not pick

pick: helper(strs, 0, 1, 5) + 1

------------------------------------------

index = 5

return 0

------------------------------------------

index = 4

pick = helper(strs, 0, 1, 5) + 1 = 1

not pick: helper(strs, 1, 1, 5)

------------------------------------------

index = 5

return 0

------------------------------------------

index = 4

pick = 1

not pick = helper(strs, 1, 1, 5) = 0

return Math.max(1, 0) = 1

------------------------------------------

index = 3

pick = helper(strs, 1, 1, 4) + 1 = 2

not pick: helper(strs, 1, 2, 4)

------------------------------------------

index = 4

"0" -> index\_str\_zeros = 1, index\_str\_ones = 0

m - 1 = 1 - 1 = 0 >= 0

n - 0 = 2 - 0 = 1 >= 0

have choice to pick / not pick

pick: helper(strs, 0, 2, 5) + 1

------------------------------------------

index = 5

return 0

------------------------------------------

index = 4

pick = helper(strs, 0, 2, 5) + 1 = 1

not pick: helper(strs, 1, 2, 5)

------------------------------------------

index = 5

return 0

------------------------------------------

index = 4

pick = 1

not pick = helper(strs, 1, 2, 5) = 0

return Math.max(1, 0) = 1

------------------------------------------

index = 3

pick = 2

not pick = helper(strs, 1, 2, 4) = 1

return Math.max(2, 1) = 2

------------------------------------------

index = 2

return helper(strs, 1, 2, 3) = 2

------------------------------------------

index = 1

pick = helper(strs, 1, 2, 2) + 1 = 3

not pick: helper(strs, 4, 3, 2)

------------------------------------------

index = 2

"111001" -> index\_str\_zeros = 2, index\_str\_ones = 4

m - 1 = 4 - 2 = 2 < 0

n - 1 = 3 - 4 = -1 < 0

no choice have to skip

return helper(strs, 4, 3, 3)

------------------------------------------

index = 3

"1" -> index\_str\_zeros = 0, index\_str\_ones = 1

m - 0 = 4 - 0 = 4 >= 0

n - 1 = 3 - 1 = 2 >= 0

have choice to pick / not pick

pick: helper(strs, 4, 2, 4) + 1

------------------------------------------

index = 4

"0" -> index\_str\_zeros = 1, index\_str\_ones = 0

m - 1 = 4 - 1 = 3 >= 0

n - 0 = 2 - 0 = 2 >= 0

have choice to pick / not pick

pick: helper(strs, 3, 2, 5) + 1

------------------------------------------

index = 5

return 0

------------------------------------------

index = 4

pick = helper(strs, 3, 2, 5) + 1 = 1

not pick: helper(strs, 4, 2, 5)

------------------------------------------

index = 5

return 0

------------------------------------------

index = 4

pick = 1

not pick = helper(strs, 4, 2, 5) = 0

return Math.max(1, 0) = 1

------------------------------------------

index = 3

pick = helper(strs, 4, 2, 4) + 1 = 2

not pick: helper(strs, 4, 3, 4)

------------------------------------------

index = 4

"0" -> index\_str\_zeros = 1, index\_str\_ones = 0

m - 1 = 4 - 1 = 3 >= 0

n - 0 = 3 - 0 = 3 >= 0

have choice to pick / not pick

pick: helper(strs, 3, 3, 5) + 1

------------------------------------------

index = 5

return 0

------------------------------------------

index = 4

pick = helper(strs, 3, 3, 5) + 1 = 1

not pick: helper(strs, 4, 3, 5)

------------------------------------------

index = 5

return 0

------------------------------------------

index = 4

pick = 1

not pick = helper(strs, 4, 3, 5) = 0

return Math.max(1, 0) = 1

------------------------------------------

index = 3

pick = 2

not pick = helper(strs, 4, 3, 4) = 1

return Math.max(2, 1) = 2

------------------------------------------

index = 2

return helper(strs, 4, 3, 3) = 2

------------------------------------------

index = 1

pick = 3

not pick = helper(strs, 4, 3, 2) = 2

return Math.max(3, 2) = 3

------------------------------------------

index = 0

pick = 3

not pick = helper(strs, 4, 3, 1) = 3

return Math.max(3, 3) = 3

------------------------------------------

Final return 3

**Solution 2:  DFS + Memoization (10 min)**

class Solution {

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

Integer[][][] memo = new Integer[strs.length + 1][m + 1][n + 1];

return helper(strs, m, n, 0, memo);

}

private int helper(String[] strs, int m, int n, int index, Integer[][][] memo) {

if(index >= strs.length) {

return 0;

}

if(memo[index][m][n] != null) {

return memo[index][m][n];

}

int index\_str\_ones = countOnes(strs[index]);

int index\_str\_zeros = strs[index].length() - index\_str\_ones;

// If remain m or n less than current element's ones or zeros we have no choice,

// not able to pick it up and have to go ahead directly to next element if exist

if(m - index\_str\_zeros < 0 || n - index\_str\_ones < 0) {

return helper(strs, m, n, index + 1, memo);

}

// Pick element at index

// Don't forget to add 1 on subset size for current pick

int pick = helper(strs, m - index\_str\_zeros, n - index\_str\_ones, index + 1, memo) + 1;

// Not pick element at index

int not\_pick = helper(strs, m, n, index + 1, memo);

return memo[index][m][n] = Math.max(pick, not\_pick);

}

private int countOnes(String str) {

int count = 0;

for(char c : str.toCharArray()) {

if(c == '1') {

count++;

}

}

return count;

}

}

Time Complexity : O(L\*M\*N), where L is the length of strs

Space Complexity : O(L\*M\*N)

**Solution 3:  DP (10 min)**

class Solution {

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

int[][][] dp = new int[strs.length + 1][m + 1][n + 1];

for(int i = strs.length - 1; i >= 0; i--) {

int index\_str\_ones = countOnes(strs[i]);

int index\_str\_zeros = strs[i].length() - index\_str\_ones;

for(int j = 0; j <= m; j++) {

for(int k = 0; k <= n; k++) {

int pick = 0;

int not\_pick = 0;

if(j >= index\_str\_zeros && k >= index\_str\_ones) {

pick = dp[i + 1][j - index\_str\_zeros][k - index\_str\_ones] + 1;

}

not\_pick = dp[i + 1][j][k];

dp[i][j][k] = Math.max(pick, not\_pick);

}

}

}

return dp[0][m][n];

}

private int countOnes(String str) {

int count = 0;

for(char c : str.toCharArray()) {

if(c == '1') {

count++;

}

}

return count;

}

}

Time Complexity : O(L\*M\*N), where L is the length of strs

Space Complexity : O(L\*M\*N)

**Solution 4:  DP + Space Optimization (10 min)**

**Wrong Solution: Define dp out side outer for loop as same as dpPrev, but dp need to initialize based on each iteration inside outer for loop, similar like L72.Edit Distance (Refer L115.Distinct Subsequences)**

class Solution {

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

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

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

for(int i = strs.length - 1; i >= 0; i--) {

int index\_str\_ones = countOnes(strs[i]);

int index\_str\_zeros = strs[i].length() - index\_str\_ones;

for(int j = 0; j <= m; j++) {

for(int k = 0; k <= n; k++) {

int pick = 0;

int not\_pick = 0;

if(j >= index\_str\_zeros && k >= index\_str\_ones) {

pick = dpPrev[j - index\_str\_zeros][k - index\_str\_ones] + 1;

}

not\_pick = dpPrev[j][k];

dp[j][k] = Math.max(pick, not\_pick);

}

}

dpPrev = dp.clone();

}

return dp[m][n];

}

private int countOnes(String str) {

int count = 0;

for(char c : str.toCharArray()) {

if(c == '1') {

count++;

}

}

return count;

}

}

**Style 1: Keep scanning j, k from 0 to m, n**

class Solution {

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

// 'dpPrev' represent the last row (strs.length th row)

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

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

for(int i = strs.length - 1; i >= 0; i--) {

int index\_str\_ones = countOnes(strs[i]);

int index\_str\_zeros = strs[i].length() - index\_str\_ones;

// Initialize dp in each iteration (start from strs.length - 1 th

// to 1st row)

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

for(int j = 0; j <= m; j++) {

for(int k = 0; k <= n; k++) {

int pick = 0;

int not\_pick = 0;

if(j >= index\_str\_zeros && k >= index\_str\_ones) {

pick = dpPrev[j - index\_str\_zeros][k - index\_str\_ones] + 1;

}

not\_pick = dpPrev[j][k];

dp[j][k] = Math.max(pick, not\_pick);

}

}

dpPrev = dp.clone();

}

return dpPrev[m][n];

}

private int countOnes(String str) {

int count = 0;

for(char c : str.toCharArray()) {

if(c == '1') {

count++;

}

}

return count;

}

}

Time Complexity : O(L\*M\*N), where L is the length of strs

Space Complexity : O(M\*N)

**Style 2: Reverse scanning j, k from m, n to 0, even optimize to only 1D array, and that's the root cause we have to reverse the scanning direction**

class Solution {

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

// 'dpPrev' represent the last row (strs.length th row), and no need 'dp'

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

for(int i = strs.length - 1; i >= 0; i--) {

int index\_str\_ones = countOnes(strs[i]);

int index\_str\_zeros = strs[i].length() - index\_str\_ones;

// Backwards scanning (Refer L115.Distinct Subsequences 1D DP explain)

// Both current j depends on its left index value as j - index\_str\_zeros,

// k depends on its left index value as k - index\_str\_ones, the untouched

// part on the right side of both j and k, and for DP formula it requires

// current status based on stable existing status, in our case here the

// stable existing status come from current status's right hand, so scanning

// backwards

for(int j = m; j >= 0; j--) {

for(int k = n; k >= 0; k--) {

int pick = 0;

int not\_pick = 0;

if(j >= index\_str\_zeros && k >= index\_str\_ones) {

pick = dpPrev[j - index\_str\_zeros][k - index\_str\_ones] + 1;

}

not\_pick = dpPrev[j][k];

dpPrev[j][k] = Math.max(pick, not\_pick);

}

}

}

return dpPrev[m][n];

}

private int countOnes(String str) {

int count = 0;

for(char c : str.toCharArray()) {

if(c == '1') {

count++;

}

}

return count;

}

}

Time Complexity : O(L\*M\*N), where L is the length of strs

Space Complexity : O(M\*N)

**Refer to**

<https://leetcode.com/problems/ones-and-zeroes/solutions/814077/dedicated-to-beginners/>

**before starting with DP one must master the art of recursion**Most of the post shows you the optimal solution, briefly explains how it works, but doesn't tell you how to arive at that solution, (this post are for begineers). it goes as "give man a fish, he can eat it whole day, tell then how to catch one, he can have it whole life"

**step1 : understand the problem**we have been given a array of BinaryStrings and two numbers, viz no of zeros and no of ones.Our task is to tell how many strings can we make by using available no of zeros and ones, such that we are able to make the maximum no of strings present in given array.

**step 2: logic building and deriving recurence relation**

**1st sub step : asking questions**

1. question that i should ask is what is the max no of strings can i make from given avaible choices.
2. ( now assume you are at a particular index on the array , these are the questions that matter )
3. if i use this current string then what all choices i am left with ? what is the max possible ans for those choices.
4. if i dont use this current string then i have other options( other indexes ) to explore , whats going to be the answer for amoung these choices.
5. Finally : if i know the answer for both the paths then i could be able to decide whether i should select this current string or not.

**2nd sub step : Arriving at recurrence relation**

at some Ith index my options is going to be

1 . if i use decide to build this string then

answerFor( next index , currAvailableZeros - currentStringZeros , currAvailableOnes - currentStringOnes )

2. if i dont decide to build this string ( assuming i will get better answer later, if i save available zeros and ones)

answerFor( next index , currAvailableZeros , currAvailableOnes )

ans\_for\_ith = maxOf ( if i build this current string , if i dont build )

**step 3: code recursive solution - (TLE)**

int sub(vector<pair<int,int>> &a, int index , int nOnes , int nZeros)

{

// BASE CASES

// 1. if we have reched the end of our array then we dont have any options left

// 2. if we have no available zeros or ones to use then also we dont have any options left

int n = a.size();

if(index == n or (nZeros ==0 and nOnes == 0)) return 0;

if(a[index].first > nOnes or a[index].second > nZeros)

return sub(a , index+1 , nOnes , nZeros);

// now at this point we have two paths to explore, we want to know that what is the answer going to be if i

// use this pair

// or if i leave this pair

int include = 1 + sub(a , index+1, nOnes - a[index].first , nZeros - a[index].second);

int exclude = sub(a , index+1 , nOnes , nZeros);

// now since we know the our answers for both of our decisions then we can choose the one which yeilds the max result

return max(include , exclude);

}

int findMaxForm(vector<string>& strs, int nZeros, int nOnes)

{

// lets make a array of pairs which is going to store no of zeros and no of ones for ith string

vector<pair<int,int>> a;

for(auto i : strs)

{

int one = 0 , zero = 0;

for(auto j : i)

(j == '1') ? one ++ : zero++;

a.push\_back({one , zero});

}

int ans = sub(a , 0 , nOnes , nZeros);

return ans;

}

**step4 : memoization-top-down**

Now we know the subProblems. lets us save them to save some computation lets maintain a DP matrix which will store the calculated values for all index and for all its possible values of m and n

int dp[601][101][101] = {};

int sub(vector<pair<int,int>> &a, int index , int nOnes , int nZeros)

{

int n = a.size();

if(index == n or (nZeros ==0 and nOnes == 0)) return 0;

// if we already know answer for this index with current nZeros , nOnes then no need to recompute return the saved answer

if(dp[index][nOnes][nZeros] != -1)

return dp[index][nOnes][nZeros];

// we dont have available no of zeros or ones to build this curent string, so we dont have any other option but to skip this index

if(a[index].first > nOnes or a[index].second > nZeros)

return dp[index][nOnes][nZeros] = sub(a , index+1 , nOnes , nZeros);

// NOTE : we are also saving corrensponding values

int include = 1 + sub(a , index+1, nOnes - a[index].first , nZeros - a[index].second);

int exclude = sub(a , index+1 , nOnes , nZeros);

// save these values and return the answer

return dp[index][nOnes][nZeros] = max(include , exclude);

}

int findMaxForm(vector<string>& strs, int nZeros, int nOnes)

{

vector<pair<int,int>> a;

memset(dp , -1 , sizeof(dp));

for(auto i : strs)

{

int one = 0 , zero = 0;

for(auto j : i)

(j == '1') ? one ++ : zero++;

a.push\_back({one , zero});

}

int ans = sub(a , 0 , nOnes , nZeros);

return ans;

}

**step 5 : bottom up DP**

now you know the base cases and subproblems , so try to come up with bottom up code yourself.

int findMaxForm(vector<string>& strs, int zeros, int ones)

{

int i,j,k,l,p=strs.size();

vector<vector<int>> dp(zeros+1 , vector<int>(ones+1));

for(auto &s:strs)

{

int currOnes = count(s.begin(),s.end(),'1');

int currZeros = s.size()-currOnes;

for(j=ones;j>=currOnes;j--)

{

for(i=zeros;i>=currZeros;i--)

{

dp[i][j] = max(dp[i][j],1+dp[i-currZeros][j-currOnes]);

}

}

}

return dp[zeros][ones];

}

**Refer to**

<https://leetcode.com/problems/ones-and-zeroes/solutions/2065992/c-detailed-explanation-w-recursion-memoziation-examples-and-well-commented/>

***Brief note about Question-***

* We have to return the size of the largest subset of strs such that there are at most m 0's and n 1's in the subset.

Let's take an example not given in the question-

Suppose our arr is arr[]: ["101", "111", "000", "01", "111110100"],

Given, m = 5 and n = 6

Now, This says that we can include at most 5 zeroes and at most 6 ones in our ans,

So, we say that we will choose "101", "111", "000", and "01"

by choosing this, we satisy our criteria of atmost 5 zeroes and 6 ones,

so the answer is 4.

One thing is to notice here is that we have to choose atmost,

means maximum we can choose for zero is 'm' and maximum we can choose for one is 'n'.

We have option to choose less than 'm' and 'n' but to maximise the count of subset.

***Generating intuition:***

* First thing that we will see is that from the given array of strings we have to choose some of its indices,such that after choosing them we will count our ones and zeros.
* So, this is kind of ***option type thing***, suppose we start traversing in our given vector, and stand on any ith index, then for that index we have a choice.
* And **what choice** is that? Choice is that we will ask a question to ourself whether to include this string in our answer or not.
* ***So, if we decide to choose this string in our answer then we will add count of ones and zeroes of this string in our answer and move ahead and if we decide not to include this string in our answer, then we will do nothing, we just move forward.***
* So, Overall we can say that we have choice for every index of array whether to include it our answer or not to include.
* And whenver we heard a term that here we have a choice like this, then we will say **recursion** will use here.
* **Why recusrion?** Recursion is because it will try out every single possibility for every string in our answer.
* Let see ahead with a example.

***Again a example for better understanding:***

We will take a very light example,

Suppose array given to us is as arr[]: ["00", "1","0", "000", "11"], Given m = 3 and n = 2

It says that we can choose atmost three zeroes (m = 3) and also we can choose atmost two ones(n = 2)

See here, we can say that we have to choose three zeroes, so what are the options or to say

what are the choices we have,

We have a option in which we will say for zeroes we will choose "000", it complete our criteria

of having atmost three zeroes and give us the count of subset as 1.

But, since we want to maximise our answer, we will say that it is better to choose "00" and "0",

it also satisfies our criteria of having atmost three zeroes, but with the count of subset as 2.

So, we saw here we will have choice for every string in our array, whether to include this in our answer or not.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Visuliaze something like this, arr[]: ["00", "1","0", "000", "11"]

/ \

/ \

whether to choose this whether to not choose this

{Same for all other strings}

And that is what we are going to implement.

***Now What?***

* Now till here we understand that ok we have option for every string in our array. Now what?
* We will ***keep a record of count ones and zeroes***.
* We say when we decide to include this string in our answer, then we add count of ones and zeroes of this string in our answer and then move to anthor index.
* But if we decide not to choose this, so we do not do any change in our counts, because we are not choosing this.
* And also, on including a string in our anwer we will see whether after including this in our answer our criteria of having atmost m zeroes and n zeroes are satisfied or not.
* If it crosses that criteria then we will do nothing we simply move ahead.
* But if it not crosses then criteria, then we will apply our option that whether to choose this in our answer.
* See commented code for more explanation.
* We will move further with first recursion and then memoziation.

***Solution - I (Recursion, TLE)-***

class Solution {

public:

// Count one and Zero function take string as parameter and count the number of ones and zeroes present in the string and return the counts.

pair<int, int> countOneAndZero(string s)

{

int one = 0, zero = 0;

for(int i = 0; i < s.length(); i++) // travel in the string

{

if(s[i] == '1') // if == '1', then add to one

one++;

else // otherwise add to zero

zero++;

}

return {one, zero};

}

int solve(int i, int one, int zero, int& maxZero, int& maxOne,

vector<string>& arr)

{

if(i >= arr.size()) // if ith index crosses the length then return 0

return 0;

// if any of the count, crosses the criteria of having maximum one

// or zero, then return 0

if(one > maxOne || zero > maxZero)

return 0;

/\* what we discused:-

for every ith index i, we have two option, whether to include it

in our answer or not, if include then add the count of

ones and zeros from that string \*/

// pair p contains, the number of ones and zeroes present in the string of ith index of vector arr.

pair<int, int> p = countOneAndZero(arr[i]);

/\* we declare three variables -

1) ans1, If adding the count of ones and zeroes at ith index in arr,

does not crosses our limit, then to include this in our answer.

2) ans2, If adding the count of ones and zeroes at ith index in arr,

does not crosses our limit, then not to include this in our answer.

3) ansWithout, If adding the count of ones and zeroes at ith index in arr, crosses our limit, then not to include this in our answer.

\*/

int ans1 = 0, ans2 = 0, ansWithout = 0;

// adding count of current index, not to cross our limit then-

if(one + p.first <= maxOne && zero + p.second <= maxZero)

{

// ans1, including it in our answer

ans1 = 1 + solve(i + 1, one + p.first, zero + p.second,

maxZero, maxOne, arr);

// not including in our answer.

ans2 = solve(i + 1, one, zero, maxZero, maxOne, arr);

}

else // if crossing limit, obviously not to take

{

ansWithout = solve(i + 1, one, zero, maxZero, maxOne, arr);

}

// and at last return the maximum of them

return max({ans1, ans2, ansWithout});

}

int findMaxForm(vector<string>& arr, int m, int n) {

// we redfine m and n as maxzero and maxOne respectively, for better clarification.

int maxZero = m;

int maxOne = n;

/\* parameters of solve-

1) First parameter is 'i', by which we move in our array of strings.

2) Second parameter is the variable who keeps record of count of ones. (we name it as one)

3) Third parameter is the variable who keeps record of count of zeroes. (we name it as zero)

4) Fourth parameter is maxZero, the maximum zeros we are allowed to take.

5) Fifth parameter is maxOne, the maximum ones we are allowed to take.

6) Last Sixth parameter is the array itself.

\*/

// return solve function

return solve(0, 0, 0, maxZero, maxOne, arr);

}

};

***Solution - II (Memoziation, Accepted)-***

* In memoziation, we make an dp array, to store some already previous computed result.
* ***How many parameter are there which are changing?***
* We will see, first index itself and then count of ones and and zeroes.
* So, we will a 3 D vector for storing results that are already previous computed.
* See, code :)

class Solution {

public:

// 3 D dp vector, as we dicused, We will give it maximum size,

//see Constraints, 1 <= strs.length <= 600 and 1 <= m, n <= 100

int dp[601][101][101];

// Count one and Zero function take string as parameter and count the number of ones and zeroes present in the string and return the counts.

pair<int, int> countOneAndZero(string s)

{

int one = 0, zero = 0;

for(int i = 0; i < s.length(); i++) // travel in the string

{

if(s[i] == '1') // if == '1', then add to one

one++;

else // otherwise add to zero

zero++;

}

return {one, zero};

}

int solve(int i, int one, int zero, int& maxZero, int& maxOne,

vector<string>& arr)

{

if(i >= arr.size()) // if ith index crosses the length then return 0

return 0;

// if any of the count, crosses the criteria of having maximum one

// or zero, then return 0

if(one > maxOne || zero > maxZero)

return 0;

// if it is already computed, then no need to do computation again,

//return from here itself

if(dp[i][one][zero] != -1)

{

return dp[i][one][zero];

}

/\* what we discused:-

for every ith index i, we have two option, whether to include it

in our answer or not, if include then add the count of

ones and zeros from that string \*/

// pair p contains, the number of ones and zeroes present in the string of ith index of vector arr.

pair<int, int> p = countOneAndZero(arr[i]);

/\* we declare three variables -

1) ans1, If adding the count of ones and zeroes at ith index in arr,

does not crosses our limit, then to include this in our answer.

2) ans2, If adding the count of ones and zeroes at ith index in arr,

does not crosses our limit, then not to include this in our answer.

3) ansWithout, If adding the count of ones and zeroes at ith index in arr, crosses our limit, then not to include this in our answer.

\*/

int ans1 = 0, ans2 = 0, ansWithout = 0;

// adding count of current index, not to cross our limit then-

if(one + p.first <= maxOne && zero + p.second <= maxZero)

{

// ans1, including it in our answer

ans1 = 1 + solve(i + 1, one + p.first, zero + p.second,

maxZero, maxOne, arr);

// not including in our answer.

ans2 = solve(i + 1, one, zero, maxZero, maxOne, arr);

}

else // if crossing limit, obviously not to take

{

ansWithout = solve(i + 1, one, zero, maxZero, maxOne, arr);

}

// and at last return the maximum of them

return dp[i][one][zero] = max({ans1, ans2, ansWithout});

}

int findMaxForm(vector<string>& arr, int m, int n) {

// we redfine m and n as maxzero and maxOne respectively, for better clarification.

int maxZero = m;

int maxOne = n;

memset(dp, -1, sizeof(dp)); // intially, putting -1 in dp

/\* parameters of solve-

1) First parameter is 'i', by which we move in our array of strings.

2) Second parameter is the variable who keeps record of count of ones. (we name it as one)

3) Third parameter is the variable who keeps record of count of zeroes. (we name it as zero)

4) Fourth parameter is maxZero, the maximum zeros we are allowed to take.

5) Fifth parameter is maxOne, the maximum ones we are allowed to take.

6) Last Sixth parameter is the array itself.

\*/

// return solve function

return solve(0, 0, 0, maxZero, maxOne, arr);

}

};

* **Note:** I am not providing bottom up soln, because every second post is that itself.
* Main thing is to come up with recursive approach and then memoize it.

**Refer to**

<https://leetcode.com/problems/ones-and-zeroes/solutions/1138589/short-easy-w-explanation-o-l-m-n-dp-solution-6-lines-similar-to-knapsack/>

For each string in the set, we have the choice to include it in the subset or leave it. For Max(strs.length) == 600, we would have 2^600, different ways of choosing and we have to find the way which maximizes the subset size. Obviously, this exponential Time complexity brute force approach won't work.

✔️ ***Solution (Dynamic Programming)***

This problem looks a lot like the knapsack just that we have two constraints here m and n instead of just W in the knapsack problem. Here, a set of items(strings) are given and we have to choose a subset satisfying given constraints. We can apply **DP** here. We can maintain a 2d dp array, where dp[i][j] will maintain the optimal solution when zeros\_limit = i & ones\_limit = j.

For each string, some number of 0s(lets call *zeros*) and 1s (lets call *ones*) are required. Obviously, if our balance of zeros and ones is less than what is required by current string, we can't choose it. But in the case where our balance of *zeros* and *ones* is greater than the required, we have two cases -

* Either take the current string into our subset. The resultant count would be 1 + optimal solution that we had when our balance was i - zeros & j - ones.
* Or leave the current string meaning the resultant count will remain the same.

For each string in strs, we will update the dp matrix as per the above two cases.

int findMaxForm(vector<string>& strs, int m, int n) {

// dp[i][j] will store Max subset size possible with zeros\_limit = i, ones\_limit = j

vector<vector<int> > dp(m + 1, vector<int>(n + 1));

for(auto& str : strs) {

// count zeros & ones frequency in current string

int zeros = count(begin(str), end(str), '0'), ones = size(str) - zeros;

// which positions of dp will be updated ?

// Only those having atleast `zeros` 0s(i >= zeros) and `ones` 1s(j >= ones)

for(int i = m; i >= zeros; i--)

for(int j = n; j >= ones; j--)

dp[i][j] = max(dp[i][j], // either leave the current string

dp[i - zeros][j - ones] + 1); // or take it by adding 1 to optimal solution of remaining balance

// at this point each dp[i][j] will store optimal value for items considered till now & having constraints i and j respectively

}

return dp[m][n];

}

***Time Complexity :*** **O(L\*m\*n)**, where L is the length of strs

***Space Complexity :*** **O(m\*n)**

📝 ***Some Extra Points*** 💡

* *Why go from i = m and j = n, i.e, bottom right of our dp to the top left ?***Ans :** For every string, we are updating the values of dp where it is possible to choose the current string. Now, we could do something like -

// wrong way

for(int i = 0; i <= m; i++)

for(int j = 0; j <= n; j++)

if(i >= zeros && j >= ones)

dp[i][j] = max(dp[i][j], dp[i - zeros][j - ones] + 1);

What this does is, it will update some values in the **top left of dp** and then we would be using these updated values in **updating some dp[i][j] in the bottom right**. But this would be wrong, since it would be like choosing the same string twice.

We could most certainly do it from bottom left as well...it's just a bit less efficient -

int findMaxForm(vector<string>& strs, int m, int n) {

vector<vector<int> > dp(m + 1, vector<int>(n + 1));

for(auto& str : strs) {

int zeros = count(begin(str), end(str), '0'), ones = size(str) - zeros;

vector<vector<int> > dp2 = dp;

for(int i = 0; i <= m; i++)

for(int j = 0; j <= n; j++)

if(i - zeros >= 0 && j - ones >= 0) // use this check or directly start from i=zeros & j=ones

dp2[i][j] = max(dp[i][j], dp[i - zeros][j - ones] + 1); // ensures we don't count a string multiple times

dp = dp2;

}

return dp[m][n];

}

*Recursive solution ?*

**Ans** : Here's the Top-down recursive approach -

vector<vector<vector<int> > >dp;

int findMaxForm(vector<string>& strs, int m, int n) {

dp.resize(size(strs), vector<vector<int> >(m + 1, vector<int>(n + 1)));

return helper(strs, m, n, 0);

}

int helper(vector<string>& strs, int m, int n, int idx){

// base condition - all items covered, then stop further recusion

if(idx == size(strs)) return 0;

// if the current case is already memoised - return it

if(dp[idx][m][n]) return dp[idx][m][n];

// count freqeuncy of zeros and ones in current string

int zeros = count(begin(strs[idx]), end(strs[idx]), '0'), ones = size(strs[idx]) - zeros;

dp[idx][m][n] = helper(strs, m, n, idx + 1); // case where current string is not chosen & move to next string

// if current string can be chosen, find optimal between choosing it and leaving it

if(m - zeros >= 0 && n - ones >= 0)

dp[idx][m][n] = max(dp[idx][m][n], 1 + helper(strs, m - zeros, n - ones, idx + 1));

return dp[idx][m][n]; // finally the optimal answer will be returned

}

The above solution gave runtime which was about 5x slower and space usage which was around 10x more than the iterative version. Also, the above solution is a **3d dp** solution and I wasn't able to form a **2d recursive solution**. If anyone is able to do it, **please comment below.**

*How to know this is a DP problem?***Ans** : You wont get that it's a dp problem right away. For me, I first thought it may be a greedy solution where I could sort the strs based on length of string and greedily choose starting from the smallest strings that satisfied the constraints. This didn't work because one constraint could be smaller than other and greedily choosing smallest string may end up using all of that contraint's balance at once. Tried some other ways of sorting based on m/n & using greedy approach but to no avail.

Finally decided to go back and see if there's a brute force way of solving it. I could observe that for each string, we had choice to select or leave it but this gave exponential TC. After making this observation, the problem seemed kind of similar to some other dp problems like coin change or knapsack problem where we have to make choice about each item - whether to take it or leave it. But this problem had 2 constrains m and n

Now, for 1-contraint dp problem like knapsack, we would do something like -

for item in items\_list:

for w in range(0, MAX\_W+1): // build optimal answers for w from [0, max\_W]

// take it or leave to get optimal answer

// now dp will have optimal answers for items considered till now

In this case, we would have to deal with both the constraints instead of one -

for \_string in strs:

for i in range(0, m+1):

for j in range(0, n+1): // build optimal answers for both constraints - for each j in [0,n] & do this for each i in [o,m]

// again take it or leave choice will be evaluated for \_string

// now dp will have optimal answers for strings considered till now

Now, the solution I presented above is analogous to optimized 1d approach in knapsack. In this, the non-optimized space approach would be to use 3d dp where dp[i][j][k] would denote optimal solution for first i items when constraints m and n are j and k respectively. This is shown in the recursive solution given above