<https://leetcode.com/problems/count-ways-to-build-good-strings/description/>

Given the integers zero, one, low, and high, we can construct a string by starting with an empty string, and then at each step perform either of the following:

Append the character '0' zero times.

Append the character '1' one times.

This can be performed any number of times.

A **good** string is a string constructed by the above process having a **length** between low and high (**inclusive**).

Return *the number of****different****good strings that can be constructed satisfying these properties.* Since the answer can be large, return it **modulo** 109 + 7.

**Example 1:**

**Input:** low = 3, high = 3, zero = 1, one = 1

**Output:** 8

**Explanation:** One possible valid good string is "011". It can be constructed as follows: "" -> "0" -> "01" -> "011". All binary strings from "000" to "111" are good strings in this example.

**Example 2:**

**Input:** low = 2, high = 3, zero = 1, one = 2

**Output:** 5

**Explanation:** The good strings are "00", "11", "000", "110", and "011".

**Constraints:**

1 <= low <= high <= 10^5

1 <= zero, one <= low

**Attempt 1: 2025-06-20**

**Solution 1: Native DFS (60 min)**

**Style 1: int return**

**Wrong Solution**

**Input: low = 2, high = 3, zero = 1, one = 2**

**Output = 3**

**Expected = 5**

class Solution {

    int MOD = (int)(1e9 + 7);

    public int countGoodStrings(int low, int high, int zero, int one) {

        return helper(0, low, high, zero, one);

    }

    private int helper(int curLen, int low, int high, int zero, int one) {

        if(curLen >= low && curLen <= high) {

            return 1;

        }

        if(curLen > high) {

            return 0;

        }

        int count = 0;

        count = (count + helper(curLen + zero, low, high, zero, one)) % MOD;

        count = (count + helper(curLen + one, low, high, zero, one)) % MOD;

        return count;

    }

}

**Corrected int return solution (TLE 14/36)**

class Solution {

    int MOD = (int)(1e9 + 7);

    public int countGoodStrings(int low, int high, int zero, int one) {

        return helper(0, low, high, zero, one);

    }

    private int helper(int curLen, int low, int high, int zero, int one) {

        // Base case: stop if length exceeds high

        if(curLen > high) {

            return 0;

        }

        // Count current string if valid

        // And NO return here (stop here), since we have to explore longer

        // valid strings that could be formed by extending the current string.

        int count = 0;

        if(curLen >= low) {

            count = (count + 1) % MOD;

        }

        // Explore both options: append zeros or ones

        count = (count + helper(curLen + zero, low, high, zero, one)) % MOD;

        count = (count + helper(curLen + one, low, high, zero, one)) % MOD;

        return count;

    }

}

Time Complexity: O(2^high)

Space Complexity: O(high)

**Style 2: void return (TLE 14/36)**

class Solution {

    int MOD = (int)(1e9 + 7);

    int count = 0;

    public int countGoodStrings(int low, int high, int zero, int one) {

        helper(0, low, high, zero, one);

        return count;

    }

    private void helper(int curLen, int low, int high, int zero, int one) {

        // Base case: stop if length exceeds high

        if(curLen > high) {

            return;

        }

        // Count current string if valid

        // And NO return here (stop here), since we have to explore longer

        // valid strings that could be formed by extending the current string.

        if(curLen >= low) {

            count = (count + 1) % MOD;

        }

        // Explore both options: append zeros or ones

        helper(curLen + zero, low, high, zero, one);

        helper(curLen + one, low, high, zero, one);

    }

}

Time Complexity: O(2^high)

Space Complexity: O(high)

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

class Solution {

    int MOD = (int)(1e9 + 7);

    public int countGoodStrings(int low, int high, int zero, int one) {

        Integer[] memo = new Integer[high + 1];

        return helper(0, low, high, zero, one, memo);

    }

    private int helper(int curLen, int low, int high, int zero, int one, Integer[] memo) {

        // Base case: stop if length exceeds high

        if(curLen > high) {

            return 0;

        }

        if(memo[curLen] != null) {

            return memo[curLen];

        }

        // Count current string if valid

        // And NO return here (stop here), since we have to explore longer

        // valid strings that could be formed by extending the current string.

        int count = 0;

        if(curLen >= low) {

            count = (count + 1) % MOD;

        }

        // Explore both options: append zeros or ones

        count = (count + helper(curLen + zero, low, high, zero, one, memo)) % MOD;

        count = (count + helper(curLen + one, low, high, zero, one, memo)) % MOD;

        return memo[curLen] = count;

    }

}

Time Complexity: O(high) - Each length is computed exactly once

Space Complexity: O(high) - For the memoization array and recursion stack

**Solution 3: DP (60 min)**

**通过本题这样一个不能在满足条件时直接返回而必须继续寻找下一个满足条件的答案的特殊例子，在 DFS 到 DP 的转换过程中寻找到了一个之前没注意到的规律，就是只有在 DFS 解法中 return 的时候才会对应到 dp 数组赋值的动作，比如本题中只有在 Base Condition 的 return 0 和最后的 return count 的时候才会对应到 dp 数组中赋值的动作**

class Solution {

    public int countGoodStrings(int low, int high, int zero, int one) {

        int MOD = (int)(1e9 + 7);

        // dp[i] = number of good strings starting from length i

        int[] dp = new int[high + 1];

        // for loop iteration direction mapping to original DFS as bottom up

        // (current good string length from high to 0)

        for(int curLen = high; curLen >= 0; curLen--) {

            // Base case (same as DFS)

            if(curLen > high) {

                // No need assign 0 again since dp array initialize as 0 already

                // but we have to skip this iteration since 'dp[curLen] = 0'

                // mapping to a 'return 0' inside original DFS solution, we have

                // to terminate this iteration immediately by using 'continue'

                // as same logic as stop the mapping recursion call in original

                // DFS solution by using 'return'

                //dp[curLen] = 0;

                continue;

            }

            int count = 0;

            // Same as original DFS solution, no 'return' in DFS then no assign

            // value to dp array

            if(curLen >= low) {

                count = (count + 1) % MOD;

            }

            // Continue explore both options: append zeros or ones

            if(curLen + zero <= high) {

                count = (count + dp[curLen + zero]) % MOD;

            }

            if(curLen + one <= high) {

                count = (count + dp[curLen + one]) % MOD;

            }

            // Now mapping to original DFS will eventually as 'return' happen, now

            // assign value to dp array happen again

            dp[curLen] = count;

        }

        return dp[0];

    }

}

Time Complexity: O(high) - Each length is computed exactly once

Space Complexity: O(high) - For the memoization array and recursion stack

**Refer to Deepseek**

**Why below solution error out when input below:**

**Input: low = 2, high = 3, zero = 1, one = 2**

**Output = 3**

**Expected = 5**

class Solution {

int MOD = (int)(1e9 + 7);

public int countGoodStrings(int low, int high, int zero, int one) {

return helper(0, low, high, zero, one);

}

private int helper(int curLen, int low, int high, int zero, int one) {

if(curLen >= low && curLen <= high) {

return 1;

}

if(curLen > high) {

return 0;

}

int count = 0;

count = (count + helper(curLen + zero, low, high, zero, one)) % MOD;

count = (count + helper(curLen + one, low, high, zero, one)) % MOD;

return count;

}

}

**The initial DFS solution had a critical flaw: it returned immediately upon reaching a valid string length (between low and high), which prevented exploring longer valid strings that could be formed by extending the current string.**

**Corrected DFS Solution**

class Solution {

int MOD = (int)(1e9 + 7);

public int countGoodStrings(int low, int high, int zero, int one) {

return helper(0, low, high, zero, one);

}

private int helper(int curLen, int low, int high, int zero, int one) {

// Base case: stop if length exceeds high

if (curLen > high) return 0;

int count = 0;

// Count current string if valid

if (curLen >= low) count = (count + 1) % MOD;

// Explore both options: append zeros or ones

count = (count + helper(curLen + zero, low, high, zero, one)) % MOD;

count = (count + helper(curLen + one, low, high, zero, one)) % MOD;

return count;

}

}

**Key Fix**

**Continue Exploration After Valid Lengths**

Instead of returning immediately when curLen is valid, we:

Count the current valid string

Continue exploring longer strings by appending more characters

**Proper Base Case**

Only stop recursion when curLen > high

**Example Walkthrough**

**Input**: low=2, high=3, zero=1, one=2

**Valid Strings**:

Length 2: "00", "11"

Length 3: "000", "011", "110"

**DFS Execution**:

Start at length 0

Two choices:

Append 1 zero → length 1

Append 2 ones → length 2

At length 1 (not valid):

Append 1 zero → length 2 (valid)

Append 2 ones → length 3 (valid)

At length 2 (valid):

Append 1 zero → length 3 (valid)

Append 2 ones → length 4 (invalid)

At length 3 (valid):

Both choices exceed high (return 0)

**Total**: 5 valid paths counted

**Time Complexity Analysis**

The time complexity of this DFS solution is **O(2^n)** where **n = high / min(zero, one)**. Here's why:

**Branching Factor**: At each recursive call, we make two new calls (adding either zero or one characters)

**Recursion Depth**: The maximum depth is high / min(zero, one) (if we always choose the smaller increment)

**Total Operations**: For depth d, we have up to 2^d operations

Worst-case when min(zero, one) = 1 → depth = high → 2^high operations

**Example**:

For low=2, high=3, zero=1, one=2:

min(zero, one) = 1

n = 3 / 1 = 3

Operations ≈ 2^3 = 8

**Space Complexity Analysis**

The space complexity is **O(n)** where **n = high / min(zero, one)**:

**Recursion Stack**: Maximum depth is high / min(zero, one)

**No Additional Storage**: Only uses stack space for recursion

**Worst-case**: When min(zero, one) = 1 → O(high) space

**Example**:

For high=100,000 and min(zero, one)=1:

Stack depth = 100,000 → Likely to cause stack overflow

**Why This Fails for Larger Inputs**

**Exponential Growth**:

For high=30 → 2^30 ≈ 1 billion operations

For high=100,000 → 2^100,000 (computationally infeasible)

**Stack Overflow**:

Deep recursion stacks will exceed JVM limits for high > 10,000

**Refer to**

[L70.Climbing Stairs](note://WEBce2fed61238e31617ae6d19bbb7f7a01)