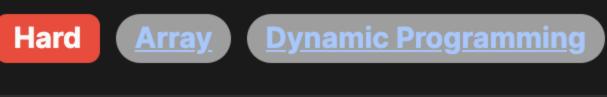
1955. Count Number of Special Subsequences



Problem Description

is considered special if it consists of a sequence containing a positive number of 0 s, then a positive number of 1 s, followed by a positive number of 2s. For example, [0, 1, 2] and [0, 0, 1, 1, 1, 2] are special subsequences, while [2, 1, 0], [1], or [0, 1, 2, 0] are not. The input array contains only integers 0, 1, and 2. The task is to return the number of such special subsequences in the array.

This problem involves finding the number of different subsequences in a given array that are considered 'special'. A subsequence

However, since the number of special subsequences could be very large, the output should be given modulo 10^9 + 7. It's important to note that a subsequence can be formed by deleting some or no elements from the array without changing the

order of the remaining elements. Two subsequences are considered different if they consist of different indices from the original array. Intuition

The intuition behind the solution is based on dynamic programming. We need to count the number of valid subsequences that end with 0, 1, and 2 separately. Since the array contains only 0s, 1s, and 2s, we can maintain a running count of the number of

Here's the logic behind counting: • When encountering a 0, we can either start a new subsequence or attach the 0 to all existing subsequences that currently end with 0. This means we double the count plus one for the new subsequence starting with just this 0.

• When encountering a 1, we can attach it either to all subsequences that end with 0 (making them end with 1 now) or to all subsequences that

subsequence must end with a 2.

valid subsequences up to the current index of the array.

- end with 1. These actions are independent: attaching to the sequences that end with 0 does not affect the ones that end with 1. So we perform addition and then double the count of subsequences that end with 1. • When encountering a 2, we use the same logic as with the 1s. A 2 can be attached to any subsequence that currently ends with a 1 (thus
- creating or extending a special sequence), or to subsequences that already end with 2. By applying this logic iteratively over the array, we can accumulate the counts. Since we want subsequences, the order of elements is crucial, and we're only going forward in the array, never backward. This forward-only approach fits perfectly with the
- dynamic programming strategy, as we can base the new counts on the previously calculated ones, ensuring that we do not count any sequence more than once.

Finally, the total number of special subsequences is the number we have accumulated that end with 2, because every special

Solution Approach The reference solution provided is an implementation of the dynamic programming approach described above. It uses an array f with three elements to keep track of the count of special subsequences that end with 0, 1, and 2, respectively. Let's dive into

how this is implemented in the solution: • We initialize the array f with zeros. However, if the first element of the input nums is a 0, we need to set f[0] to 1, since that represents a

subsequence starting with 0. • We iterate through the input nums from the second element onwards, applying the logic based on the value present in nums [i].

one for the new sequence that starts with this 0. The % mod operation ensures we stay within the bounds of the defined modulo. • Upon encountering a 1, we update f[1] to (f[0] + 2 * f[1]) % mod. This takes into account all the subsequences that can be formed by appending the 1 to existing subsequences ending with 0, as well as doubling the subsequences that already end with 1. • Likewise, when a 2 is found, f[2] gets updated with (f[1] + 2 * f[2]) % mod. This adds all sequences ending with 1 to those ending with

• When we find a 0 in nums, f[0] is updated to (2 * f[0] + 1) % mod. This captures the doubling of existing sequences ending in 0 and adds

- 2, and also doubles existing subsequences that end with 2. The iteration continues for all elements of nums. • Eventually, we return the value in f[2] because it holds the count of all special subsequences which properly end with a 2.
- By only using a fixed-size array, the solution has an O(n) time complexity, where n is the length of the input array nums, and space complexity, as the size of the f array is constant and does not depend on the input size.
- The code simply follows these steps, applying the update formulas for each element of nums. It's important to note that this solution strategy takes advantage of the property that the final special subsequence must end

special subsequences. **Example Walkthrough**

Let's walk through a small example to illustrate the solution approach. Suppose we have the following input array: nums = [0, 1,

0, 2, 1]. We need to find the number of special subsequences using dynamic programming.

new subsequence starting with this 0. Therefore, f = [1, 0, 0].

1, 0, 2, 1], the number of special subsequences is 1.

def countSpecialSubsequences(self, nums: List[int]) -> int:

Define a large number mod for taking modulus to prevent overflow

Initialize count arrays for subsequences ending with 0, 1, and 2

count and add 1 for the new subsequence.

For a 0, we can either append it to an existing subsequence of 0s

or start a new subsequence with this 0. So we double the existing

For a 2, we can append it to all existing subsequences of 2s and 1s.

count_subsequences[2] = $(count_subsequences[1] + 2 * count_subsequences[2]) % MOD$

Sequences of 1s become new subsequences of 2s when a 2 is appended.

count_subsequences[0] = (2 * count_subsequences[0] + 1) % MOD

with a 2 to calculate the result. The modulo operation is used to avoid integer overflow due to the possible large number of

We initialize our array f with zeros: f = [0, 0, 0]. This array will hold counts of special subsequences that end with 0, 1, and 2. We come across the first element, 0. According to our strategy, we update f[0] by doubling the count and adding 1 for a

o ending sequences and doubling the 1 ending sequences. As we had 1 subsequence ending with o, we add it to o (existing count of 1 ending sequences) and double the 0. Now, f = [1, 1, 0]. The third element is 0 again. We update f[0] to 2 * f[0] + 1, resulting in f[0] = 3. So, f = [3, 1, 0]. This accounts for

the subsequence starting with the second 0 and doubling the count of existing 0 ending sequences.

The next element is 1. We update f[1] to account for all subsequences that can be formed by appending this 1 to existing

f[1] + 2 * f[2], which yields a 1. So, f = [3, 1, 1]. The last element is 1. We update f[1] with f[0] (sequences that get 1 appended) plus double the existing 1 ending sequences. So f[1] = 3 + 2 * 1 = 5, and f becomes [3, 5, 1].

After this iteration, f[2] contains the total count of special subsequences ending with 2, which is 1. Therefore, for nums = [0,

Now we encounter a 2. We update f[2] by taking f[1] followed by a 2 and doubling existing 2 ending sequences: f[2] =

For the result, we use modular arithmetic with $mod = 10^9 + 7$ to avoid large numbers that can result from the calculations, ensuring numbers are within bounds of integer values typically used in programming problems.

Handle the case for the first element # If it's 0, we have one subsequence starting with 0 count_subsequences[0] = int(nums[0] == 0) # Iterate through the elements starting from the second one

For a 1, it can be appended to all existing subsequences of 1s and 0s. # For subsequences of 0s, we can create new subsequences of 1s (hence f[0]). count_subsequences[1] = (count_subsequences[0] + 2 * count_subsequences[1]) % MOD

Solution Implementation

MOD = 10**9 + 7

count subsequences = [0] * 3

for i in range(1, len(nums)):

num = nums[i]

if num == 0:

elif num == 1:

else: # num == 2

from typing import List

Python

class Solution:

```
# Return the count of subsequences that end with 2, as these are the complete sequences of 0->1->2
        return count_subsequences[2]
Java
class Solution {
    // Method to count special subsequences in an array
    public int countSpecialSubsequences(int[] nums) {
        // Initialize the modulo constant (as per modulo 10^9 + 7)
        final int MODULO = 1000000007;
        // Length of the input array
        int n = nums.length;
        // Arrav to store counts for each number 0, 1, and 2
        int[] counts = new int[3];
        // If the first element of nums is 0, then there is one such subsequence
        counts[0] = nums[0] == 0 ? 1 : 0;
        // Iterate over the array starting from the second element
        for (int i = 1; i < n; ++i) {
            if (nums[i] == 0) {
                // If we find a 0, then we double the count of subsequences
                // ending with 0, and add one for the subsequence consisting of just this 0
                counts[0] = (2 * counts[0] % MODULO + 1) % MODULO;
            } else if (nums[i] == 1) {
                // If we find a 1, count of subsequences ending with 1 is
                // increased by count of subsequences ending with 0 and twice those ending with 1
                counts[1] = (counts[0] + 2 * counts[1] % MODULO) % MODULO;
            } else { // nums[i] == 2
                // If we find a 2. count of subsequences ending with 2 is
                // increased by count of subsequences ending with 1 and twice those ending with 2
                counts[2] = (counts[1] + 2 * counts[2] % MODULO) % MODULO;
        // Return the count of special subsequences that end with 2
        return counts[2];
```

};

TypeScript

class Solution {

int countSpecialSubsequences(vector<int>& nums) {

// Calculate the size of the input vector

// Iterate over the rest of the numbers

const int MOD = 1e9 + 7;

int n = nums.size();

int frequency $[3] = \{0\};$

frequency [0] = nums [0] == 0;

for (int i = 1; i < n; ++i) {

} else if (nums[i] == 1) {

} else if (nums[i] == 2) {

function countSpecialSubsequences(nums: number[]): number {

if (nums[i] == 0) {

return frequency[2];

// Initialize the modulo constant for large number arithmetic

// Initialize a frequency array to count subsequences ending with 0, 1, 2 respectively

// If the first number is 0. then we have one subsequence starting with 0

frequency[1] = (frequency[0] + 2 * frequency[1] % MOD) % MOD;

frequency [2] = (frequency [1] + 2 * frequency [2] % MOD) % MOD;

// Return the count of subsequences ending with 2, which represents all valid special subsequences

frequency [0] = (2 * frequency [0] % MOD + 1) % MOD;

const mod = 1e9 + 7; // Define the modulus to prevent integer overflow

Iterate through the elements starting from the second one

count and add 1 for the new subsequence.

For a 0, we can either append it to an existing subsequence of 0s

or start a new subsequence with this 0. So we double the existing

For a 1, it can be appended to all existing subsequences of 1s and 0s.

For a 2, we can append it to all existing subsequences of 2s and 1s.

Sequences of 1s become new subsequences of 2s when a 2 is appended.

For subsequences of 0s, we can create new subsequences of 1s (hence f[0]).

count_subsequences[1] = $(count_subsequences[0] + 2 * count_subsequences[1]) % MOD$

count_subsequences[2] = $(count_subsequences[1] + 2 * count_subsequences[2]) % MOD$

count_subsequences[0] = (2 * count_subsequences[0] + 1) % MOD

public:

```
const lengthOfNums = nums.length;
    const counts = [0, 0, 0]; // [countOfZeros, countOfOnes, countOfTwos]
    // Initialize the count of subsequences starting with 0, if first element is 0
    counts[0] = nums[0] === 0 ? 1 : 0;
    // Loop through the numbers in the array
    for (let i = 1; i < length0fNums; ++i) {</pre>
        if (nums[i] === 0) {
            // If the current element is 0, it can either form a new subsequence
            // or get attached to existing subsequences of zeros
            counts[0] = (2 * counts[0] + 1) % mod;
        } else if (nums[i] === 1) {
            // If the current element is 1, it can either start a new subsequence after
            // every subsequence of zeros or get attached to existing subsequences of ones
            counts[1] = (counts[0] + 2 * counts[1]) % mod;
        } else if (nums[i] === 2) {
            // If the current element is 2, it can either start a new subsequence after
            // every subsequence of ones or get attached to existing subsequences of twos
            counts[2] = (counts[1] + 2 * counts[2]) % mod;
    // Return the total count of special subsequences ending with 2
    return counts[2];
from typing import List
class Solution:
    def countSpecialSubsequences(self. nums: List[int]) -> int:
       # Define a large number mod for taking modulus to prevent overflow
        MOD = 10**9 + 7
       # Initialize count arrays for subsequences ending with 0, 1, and 2
        count_subsequences = [0] * 3
       # Handle the case for the first element
       # If it's 0, we have one subsequence starting with 0
        count_subsequences[0] = int(nums[0] == 0)
```

// For a new 0, double the existing subsequences and add 1 for the new subsequence starting with this 0

// For a new 1, we can append it to all the subsequences of 0's and also double the existing subsequences of 1's

// For a new 2, we can append it to all the subsequences of 1's and also double the existing subsequences of 2's

Return the count of subsequences that end with 2, as these are the complete sequences of 0->1->2return count_subsequences[2]

Time and Space Complexity

for i in range(1, len(nums)):

num = nums[i]

if num == 0:

elif num == 1:

else: # num == 2

The time complexity of the given code is O(n), where n is the length of the input list nums. This is because there is a single loop that iterates over the list nums once, and within that loop, it performs a constant amount of work for updating the list f.

The space complexity of the given code is 0(1) since the space used does not depend on the size of the input list nums. Only a fixed-size list f of length 3 is used for the entire algorithm, regardless of the input size.