2143. Choose Numbers From Two Arrays in Range

Hard · Array · Dynamic Programming · Leetcode Link

Problem Description

The problem provides two arrays `nums1` and `nums2` of the same length `n`. The task is to find the number of different balanced ranges within these arrays. A balanced range `l, r` means that for every index `i` within this range, you can choose either `nums1[i]` or `nums2[i]` such that the sum of the selected elements from `nums1` is equal to the sum of the selected elements from `nums2`.

Balanced ranges `[l1, r1]` and `[l2, r2]` are considered different if either the starting points `l1` and `l2` differ, the ending points `r1` and `r2` differ, or there is at least one index `i` where `nums1[i]` is chosen in the first range, and `nums2[i]` is chosen in the second range, or the other way around.

The solution must return the count of these distinctive balanced ranges modulo `10^9 + 7`.

Intuition

The solution is based on dynamic programming. The primary intuition is to track the difference between sums of selected elements from `nums1` and `nums2` at each index `i` when considering all possible subranges ending at that index. Essentially, you want to know how many ways you can achieve a particular sum difference up to a certain point, which in turn will help you decide how many balanced subranges you can form.

Here's how you arrive at the solution:

1. Create a list of lists `f`, where `f[i][j]` stores the number of ways to achieve a sum difference of `j - s2` using subranges that end at index `i`. `s2` is the total sum of `nums2`, and this acts as a balance point to avoid negative indices in `f` due to potential negative differences.
2. Iterate through each index `i` of the arrays, and for each `i`, consider adding the current value `a` from `nums1` or subtracting the current value `b` from `nums2` to all previously computed sum differences to update the number of ways to achieve new differences at `i` based on those at index `i - 1`. That is, `f[i][j]` is updated by adding `f[i - 1][j - a]` and `f[i - 1][j + b]` considering the bounds where these indices do not go beyond the size of `f`.
3. Each time, keep track of the number of balanced subranges, which essentially corresponds to the value of `f[i][s2]` because a sum difference of `0` indicates that the sums from both `nums1` and `nums2` are equal.
4. The final answer is the accumulated count of balanced subranges modulo `10^9 + 7` to handle the large numbers as mentioned in the problem.

This approach efficiently uses the concept of prefix sums and dynamic programming to solve the problem in polynomial time.

Solution Approach

The implementation uses dynamic programming to keep track of all the possible sum differences between `nums1` and `nums2` as you iterate over them. Here is a detailed walkthrough:

1. **Initialization**: Two sums, `s1` and `s2`, are calculated to represent the total sums of `nums1` and `nums2`, respectively. A 2D list `f` with dimensions `[n][s1 + s2 + 1]` is created. This list will store the number of ways to obtain a sum difference at various `j` points, for each `i`. The `+1` accommodates zero difference.
2. **Calculating Ways for Differences**: As we iterate through `nums1` and `nums2` with index `i`, we increase `f[i][a + s2]` and `f[i][-b + s2]` by `1`. This reflects the fact that selecting `a` from `nums1` or `b` from `nums2` at the current index contributes to one way of making the sum difference of `a - b` (indexed from `-b + s2` to `a + s2` to shift the negative range).
3. **Updating Counts**: If `i` is greater than `0`, we have previous states to consider. The dynamic programming aspect comes in:
   - We update `f[i][j]` by adding the number of ways to achieve a difference of `j-a` from the previous index `i-1` to the number of ways to obtain `j` at `i`, ensuring that `j` is large enough (`j >= a`).
   - We also add the number of ways to achieve a difference of `j+b` from index `i-1` to the number of ways to obtain `j` at `i`, making sure we don't exceed the range of sum differences (`j+b < s1 + s2 + 1`).

   Here, `j` is the index representing the possible sum differences, `a` is the element from `nums1`, and `b` is the element from `nums2`.
4. **Counting Balanced Ranges**: The `f[i][s2]` entry contains the number of ways to have a zero sum difference up to index `i`, which corresponds to a balanced range. We add `f[i][s2]` to `ans`, the accumulated total of such balanced ranges, and apply `% mod` to ensure the result stays within the required modulo.
5. **Return Result**: The variable `ans` stores the final count and is returned to represent the number of different balanced ranges (modulo `10^9 + 7`).

This approach uses a dynamic table `f` and iterates through each element of `nums1` and `nums2` once, updating the counts of sum differences as it goes. The table `f` stores intermediate results that are re-used, which is a classic feature of dynamic programming, and it uses the modulo operator to manage large numbers efficiently.

Example Walkthrough

Let's consider the arrays `nums1 = [1,2,3]` and `nums2 = [2,1,2]`, and walk through the described solution approach.

Initial Setup: First, we calculate the total sums.

- `s1 = 1 + 2 + 3 = 6`
- `s2 = 2 + 1 + 2 = 5`

We initialize the 2D list `f` with dimensions `[3][6 + 5 + 1]` or `[3][12]`, as we have `3` elements and the sum difference can range from `-5` to `6`. We index `f` from `0`, so the actual index for a zero sum difference is `5`, which is `s2`.

Calculating Ways for Differences:

1. At index `0`, with `nums1[0] = 1` and `nums2[0] = 2`:
   - `f[0][1 + 5]` or `f[0][6]` represents choosing `1` from `nums1` and increases by `1`.
   - `f[0][-2 + 5]` or `f[0][3]` represents choosing `2` from `nums2` and increases by `1`.
2. At index `1`, with `nums1[1] = 2` and `nums2[1] = 1`:
   - `f[1][2 + 5]` or `f[1][7]` represents choosing `2` from `nums1`. Since `i > 0`, we add the values from `f[0][7-2]` (or `f[0][5]`, which is currently `0`) and `f[0][7+1]` (or `f[0][8]`, which does not exist and is considered `0`). Hence, `f[1][7]` increases by `1`.
   - `f[1][-1 + 5]` or `f[1][4]` represents choosing `1` from `nums2` and we perform the same additions as above, so `f[1][4]` increases by `1`.
3. At index `2`, with `nums1[2] = 3` and `nums2[2] = 2`:
   - We update `f[2][3 + 5]` or `f[2][8]` and `f[2][-2 + 5]` or `f[2][3]` similarly.

Updating Counts:

- For every `j` from `0` to `11` (which corresponds to the range of `-5` to `6`), update `f[i][j]` by adding the values from the last index `i - 1` with the differences of `j - nums1[i]` and `j + nums2[i]`, if those indices are valid.

Counting Balanced Ranges:

- For each index `i`, we add `f[i][5]` to `ans`, because `f[i][5]` represents the count of balanced ranges up to index `i`. For instance, `f[2][5]` will tell us the number of ways we can have a balanced subrange ending at index `2`.

Return Result:

- After these steps, `ans`, now containing the total count of balanced ranges, is returned as the answer modulo `10^9 + 7`.

Plugging in the numbers:

1. After index `0`, `f[0][6] = 1` and `f[0][3] = 1`. No balanced range yet.
2. After index `1`, `f[1][7] = 1` and `f[1][4] = 1`. No balanced range yet.
3. After index `2`, `f[2][8]` and `f[2][3]` are updated. We find `f[2][5] = 1` indicating one balanced range `[0,2]`.

Our `ans` would be `1` modulo `10^9 + 7`, which is just `1`, since we found one balanced range. This would be the final returned value.

Python Solution

```python
from typing import List

class Solution:
    def countSubranges(self, nums1: List[int], nums2: List[int]) -> int:
        length = len(nums1)  # store the length of nums1 and nums2, which should be the same
        sum1, sum2 = sum(nums1), sum(nums2)  # calculate the sum of the elements in nums1 and nums2
        # create a 2D list to keep track of counts while avoiding index-out-of-range errors
        counts = [[0] * (sum1 + sum2 + 1) for _ in range(length)]
        total_count = 0  # initialize the result to accumulate the total count of valid subranges
        modulo = 10**9 + 7  # the value for modulo operation to avoid large integers

        # iterate through both lists in parallel using enumerate to get both index and elements
        for i, (num1, num2) in enumerate(zip(nums1, nums2)):
            counts[i][num1 + sum2] += 1  # increment the count where the first element is picked from nums1
            counts[i][-num2 + sum2] += 1  # increment the count where the first element is picked from nums2

            # if not on the first index, update the counts array based on previous counts
            if i:
                for j in range(sum1 + sum2 + 1):
                    if j >= num1:
                        counts[i][j] = (counts[i][j] + counts[i - 1][j - num1]) % modulo
                    if j + num2 < sum1 + sum2 + 1:
                        counts[i][j] = (counts[i][j] + counts[i - 1][j + num2]) % modulo

            # update the total count for subranges that sum up to zero difference
            total_count = (total_count + counts[i][sum2]) % modulo

        return total_count  # return the total count of valid subranges

# The function countSubranges calculates the number of subranges where the sum of selected elements from nums1
# equals the sum of selected elements from nums2. It modifies the subproblem's state space to make it
# solvable using dynamic programming, ensuring that each choice at every step is either to include
# an element from nums1 or an element from nums2.
```

Java Solution

```java
class Solution {
    public int countSubranges(int[] nums1, int[] nums2) {
        int n = nums1.length; // Get the length of the input arrays.
        int sumNums1 = Arrays.stream(nums1).sum(); // Sum of all elements in nums1.
        int sumNums2 = Arrays.stream(nums2).sum(); // Sum of all elements in nums2.

        // Create a 2D array to store the number of ways to form subranges.
        int[][] dp = new int[n][sumNums1 + sumNums2 + 1];
        int answer = 0; // Initialize the answer variable to store the total count of subranges.
        final int MOD = (int) 1e9 + 7; // Define the modulo value.

        // Iterate through each element in both arrays.
        for (int i = 0; i < n; ++i) {
            int num1 = nums1[i], num2 = nums2[i]; // Get the current elements from both arrays.
            dp[i][num1 + sumNums2]++; // Increment the number of ways to achieve this sum with only the current element from nums1.
            dp[i][-num2 + sumNums2]++; // Increment the number of ways to achieve this sum with only the current element from nums2.

            // If not in the first iteration, update the dp array based on previous subranges.
            if (i > 0) {
                for (int j = 0; j <= sumNums1 + sumNums2; ++j) {
                    if (j >= num1) {
                        dp[i][j] = (dp[i][j] + dp[i - 1][j - num1]) % MOD; // Add ways to achieve this sum including the current numb
                    }
                    if (j + num2 <= sumNums1 + sumNums2) {
                        dp[i][j] = (dp[i][j] + dp[i - 1][j + num2]) % MOD; // Add ways to achieve this sum including the current numb
                    }
                }
            }
            answer = (answer + dp[i][sumNums2]) % MOD; // Update the answer with the number of ways to achieve zero sum difference.
        }
        return answer; // Return the total count of subranges with zero sum difference.
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <numeric>
#include <cstring>

class Solution {
public:
    int countSubranges(vector<int>& nums1, vector<int>& nums2) {
        int n = nums1.size(); // size of the input arrays
        int sum1 = accumulate(nums1.begin(), nums1.end(), 0); // sum of nums1
        int sum2 = accumulate(nums2.begin(), nums2.end(), 0); // sum of nums2

        // We'll use a dynamic programming array 'dp' to store the number of ways
        // to get a sum taking first (i+1) elements where the sum is offset by sum2
        // to handle negative sums.
        int dp[n][sum1 + sum2 + 1];
        memset(dp, 0, sizeof(dp)); // initialize dp array to 0

        int ans = 0; // this will hold the final answer
        const int mod = 1e9 + 7; // modulo value for the answer

        // Calculate the number of subranges for each element
        for (int i = 0; i < n; ++i) {
            int a = nums1[i], b = nums2[i]; // Current elements from both arrays
            dp[i][a + sum2]++; // If we take nums1[i], add to count
            dp[i][-b + sum2]++; // If we take nums2[i], add to count

            // Update the 'dp' array for the rest of the possible sums
            if (i > 0) { // we skip the first element because there's nothing to accumulate from
                for (int j = 0; j <= sum1 + sum2; ++j) {
                    if (j >= a) {
                        // Include the current nums1[i] in the subrange and add the count
                        // from the previous subrange sum without current nums1[i]
                        dp[i][j] = (dp[i][j] + dp[i - 1][j - a]) % mod;
                    }
                    if (j + b <= sum1 + sum2) {
                        // Include the current nums2[i] in the subrange and add the count
                        // from the previous subrange sum without current nums2[i]
                        dp[i][j] = (dp[i][j] + dp[i - 1][j + b]) % mod;
                    }
                }
            }

            // Sum up the ways to achieve sum2 (offset sum is 0) for the current element
            ans = (ans + dp[i][sum2]) % mod;
        }

        return ans; // Return the total number of valid subranges
    }
};
```

Typescript Solution

```typescript
function countSubranges(nums1: number[], nums2: number[]): number {
    const lengthOfNums = nums1.length;
    const sumOfNums1 = nums1.reduce((total, current) => total + current, 0);
    const sumOfNums2 = nums2.reduce((total, current) => total + current, 0);
    // Initialize dynamic programming table to store intermediate results
    const dpTable: number[][] = Array(lengthOfNums)
        .fill(0)
        .map(() => Array(sumOfNums1 + sumOfNums2 + 1).fill(0));
    const modulo = 1e9 + 7; // The modulo value to ensure results within integer limits
    let countOfSubranges = 0; // Variable to keep the final count of subranges

    // Iterate over each pair of elements from nums1 and nums2
    for (let i = 0; i < lengthOfNums; ++i) {
        const valueFromNums1 = nums1[i];
        const valueFromNums2 = nums2[i];
        // Increase the count for the subrange that only includes current element
        dpTable[i][valueFromNums1 + sumOfNums2]++;
        dpTable[i][-valueFromNums2 + sumOfNums2]++;

        // If current index is not the first, calculate the count of subranges that end at the current index
        if (i > 0) {
            for (let j = 0; j <= sumOfNums1 + sumOfNums2; ++j) {
                // If subrange can be extended by adding value from nums1
                if (j >= valueFromNums1) {
                    dpTable[i][j] = (dpTable[i][j] + dpTable[i - 1][j - valueFromNums1]) % modulo;
                }
                // If subrange can be extended by subtracting value from nums2
                if (j + valueFromNums2 <= sumOfNums1 + sumOfNums2) {
                    dpTable[i][j] = (dpTable[i][j] + dpTable[i - 1][j + valueFromNums2]) % modulo;
                }
            }
        }
        // Add the count of balanced subranges (where sum of nums1 elements equals sum of nums2 elements) to the answer
        countOfSubranges = (countOfSubranges + dpTable[i][sumOfNums2]) % modulo;
    }

    // Return the total count of subranges found
    return countOfSubranges;
}
```

Time and Space Complexity

The given Python code aims to count the number of subranges in two arrays, `nums1` and `nums2`, where for each subrange `(i, j)`, the sum from `i` to `j` in `nums1` is equal to the sum from `i` to `j` in `nums2`. The algorithm uses dynamic programming to keep track of the possible sums.

Time Complexity

To analyze the time complexity, we consider the number of operations performed:

- The algorithm iterates over each pair `(a, b)` from `nums1` and `nums2`.
- Inside the outer loop that iterates over `n`, where `n` is the length of `nums1` (or `nums2`), there is an inner loop that runs from `0` to `s1 + s2`, where `s1` is the sum of all elements in `nums1` and `s2` is the sum of all elements in `nums2`. This means that the inner loop runs for `O(s1 + s2)` iterations for each `i`.
- The inner loop operations consist of a constant number of arithmetic operations, each having a time complexity of `O(1)`.

Combining this information, the total time complexity is `O(n * (s1 + s2))`, as there are `n` iterations in the outer loop and `O(s1 + s2)` operations for each iteration in the inner loop.

Space Complexity

For space complexity, we consider the storage used:

- The algorithm allocates a 2D list `f` with `n` rows and `s1 + s2 + 1` columns, where `n` is the length of the arrays and `s1 + s2` is the sum of the elements in `nums1` and `nums2`. Therefore, the space required for this list is `O(n * (s1 + s2))`.
- Other variables used (`n`, `s1`, `s2`, `ans`, `mod`, `a`, `b`, `i`, `j`) require constant space, hence `O(1)`.

This results in a total space complexity of `O(n * (s1 + s2))`, dominated by the space required for the 2D list `f`.