2300. Successful Pairs of Spells and Potions

Medium Array Two Pointers Binary Search Sorting Leetcode Link

Problem Description

In this problem, you are provided with two arrays called `spells` and `potions`. The `spells` array contains elements representing the strength of each spell, and similarly, the `potions` array contains elements representing the strength of each potion. The task is to determine for each spell, how many potions can successfully combine with it to achieve or exceed a given `success` threshold. A spell and potion pair is deemed successful if the product (multiplication) of their strengths is at least equal to the `success` value provided. You should return an array where each value corresponds to the number of successful potion combinations for each spell in the `spells` array.

To clarify with an example, suppose `spells = [10, 20]`, `potions = [5, 8, 10]`, and `success = 100`. To find successful pairs, you multiply each spell with each potion:

- For the first spell (strength 10), it pairs successfully with two potions (potions of strength 10 and 8) since both products are at least 100.
- For the second spell (strength 20), it pairs successfully with all three potions, as all products will be 100 or greater. The result array will be `[2, 3]` as the first spell has 2 successful combinations, and the second has 3.

Intuition

The goal is to find an efficient way to pair spells with potions such that the product of their strengths is at least the success target. A brute-force approach would be to try all possible pairs, but this can be inefficient, especially with large arrays.

Instead, by first sorting the `potions` array, we can take advantage of binary search to quickly find the number of successful potions for each spell. Binary search operates by repeatedly dividing the search interval in half, which is much faster than scanning every element.

Once the `potions` array is sorted, for each spell, we want to find the position where the potion strength is just enough or more to meet the `success` target when paired with the spell. We need to find the smallest potion that, when multiplied by the spell's strength, equals or exceeds the `success` threshold. All potions to the right in the sorted array are guaranteed to also form successful pairs because they are equal or stronger.

The solution leverages the `bisect_left` function from Python's `bisect` module, which uses binary search to find the insertion point for a given element to maintain sorted order. In this context, it finds the first index in `potions` where the potion strength is sufficient for the `success` criteria with a given spell. We then subtract this index from the total number of potions (`m`) to get the count of successful potions for that spell.

For each spell in `spells`, we apply this logic and generate the final result array.

Solution Approach

The solution implements a binary search mechanism to optimize the process of finding successful spell-potion pairs. Here's a step-wise explanation of the approach used in the solution:

1. **Sorting the Potions**: Begin by sorting the `potions` array. This is crucial for binary search to work since binary search requires a sorted array to function correctly. Sorting is done in ascending order.
2. **Using Binary Search**: For each spell's strength value, we apply a binary search to determine the number of potions that, when multiplied by this spell's strength, will at least be equal to the `success` value. This is done by finding the leftmost (smallest) potion index that can achieve the required target when combined with the current spell.
3. **Calculating Successful Pairs**:
    - We use the `bisect_left` function which finds the index at which we could insert the value `success / spell_strength` into `potions` to maintain sorted order.
    - The value we are searching for is calculated by `success / spell_strength` because we want to find the potion strength that, at the minimum, when multiplied by the spell's strength, is equal to `success`.
    - Since `potions` is sorted, every potion at and beyond the index returned by `bisect_left` would result in a successful pair when combined with the current spell.
4. **Storing Results**:
    - For each spell, we calculate `m - index` to find out the number of successful potions, where `m` is the total number of potions and `index` is the position returned by `bisect_left`.
    - This operation gives us the count because all elements from the position `index` to the end of the potions array will be successful pairs due to the sorted property of the array.
5. **Generating the Output**: Finally, we return a list comprehension that iterates over every spell in `spells`, applies the above logic using `bisect_left`, and calculates the number of successful pairs for that spell.

Here's the critical code snippet with explanations for clarity:

```python
potions.sort() # Step 1: Sorting the 'potions' array.
m = len(potions) # Storing the length of the 'potions' array.
# Step 2, 3, and 4: List comprehension iterating over 'spells'.
return [m - bisect_left(potions, success / v) for v in spells]
```

Through sorting and binary searching, this algorithm achieves a complexity of `O(n log m)`, where `n` is the number of spells, and `m` is the number of potions. Sorting takes `O(m log m)`, and then each binary search operation takes `O(log m)`, with the linear iteration over `spells` representing the `n` in the complexity.

Example Walkthrough

Let's walk through a small example based on the solution approach given above.

Consider we have `spells = [15, 10]`, `potions = [1, 5, 20, 8]`, and `success = 120`. We aim to determine for each spell, how many potions achieve or exceed the `success` threshold when multiplied by the spell.

1. **Sorting the Potions**: The first step is sorting the `potions` array. Before: `[1, 5, 20, 8]` After sorting: `[1, 5, 8, 20]`
2. **Using Binary Search**: We then apply binary search to find the count of potions that can pair with a spell to achieve the `success`.
    - For spell `15`: We search for the smallest potion that when multiplied is at least `120`. That potion should be `120 / 15 = 8`.
    - For spell `10`: We need a potion of at least `120 / 10 = 12`.
3. **Calculating Successful Pairs**:
    - For spell `15`: Using `bisect_left`, we find the index where `8` could fit in the sorted `potions` array `[1, 5, 8, 20]`, which is index `2`.
    - For spell `10`: Using `bisect_left`, we find the index where `12` could fit, which is after `8` and before `20`. Hence, the index would be `3`.
4. **Storing Results**:
    - For spell `15`: `m - index` is `4 - 2 = 2`. So, there are 2 successful potion combinations for the spell `15`.
    - For spell `10`: `m - index` is `4 - 3 = 1`. Therefore, there is 1 successful potion combination for the spell `10`.
5. **Generating the Output**: According to the steps described in the solution approach, we create a list with the counts of successful combinations for each spell. The result for our example will be `[2, 1]`.

For better understanding, here's how the binary search part of the code would operate in a step-by-step manner:

- Spell is `15`: `bisect_left([1, 5, 8, 20], 120/15)`
- It finds the position of `8` because `8 * 15` is exactly `120`.
- `index = 2`, total potions `m = 4`, successful combinations: `4 - 2 = 2`.
- Spell is `10`: `bisect_left([1, 5, 8, 20], 120/10)`
- It passes over `8` (since `8 * 10` is less than `120`) and settles before `20`.
- `index = 3`, successful combinations: `4 - 3 = 1`.

Thus, the final returned value for this particular example with the `spells` and `potions` given would be `[2, 1]`. Each spell has a corresponding count of potions that, when multiplied, meet or exceed the `success` threshold.

Python Solution

```python
from bisect import bisect_left
from typing import List

class Solution:
    def successful_pairs(self, spells: List[int], potions: List[int], success: int) -> List[int]:
        """
        Find the number of potions for each spell that when combined
        result in a product equal to or greater than the success threshold.

        Args:
        spells: List of integers representing the strength of spells.
        potions: List of integers representing the volume of potions.
        success: An integer representing the minimum success threshold for a spell-potion combination.

        Returns:
        List containing the count of successful potion combinations for each spell.
        """
        # Sort the potions list in ascending order for binary searching
        potions.sort()

        # Length of the potions list
        num_potions = len(potions)

        # Using list comprehension to generate the list of counts for successful combinations
        # For each spell strength 'spell_strength' in 'spells', we find the index of the first potion
        # in the sorted 'potions' list that meets or exceeds the 'success' threshold when combined with the spell.
        # The count of successful pairings for each spell is then the total number of potions
        # minus this index, giving the number of potions that meet or exceed the product threshold.
        return [num_potions - bisect_left(potions, success / spell_strength) for spell_strength in spells]

```

Java Solution

```java
import java.util.Arrays; // Ensure to import the necessary Arrays class for sorting.

class Solution {

    // This function determines the number of successful pairs of spells and potions.
    public int[] successfulPairs(int[] spells, int[] potions, long successThreshold) {
        Arrays.sort(potions); // Sort the potions array for binary search.
        int nSpells = spells.length; // Number of spells.
        int nPotions = potions.length; // Number of potions.
        int[] successfulPairs = new int[nSpells]; // Array to store the answer.

        // Iterate over each spell.
        for (int i = 0; i < nSpells; ++i) {
            int left = 0, right = nPotions; // Set search boundaries for binary search.

            // Binary search to find the first potion that results in a successful pair.
            while (left < right) {
                int mid = (left + right) / 2; // Calculate the mid index.
                // Check if the current spell and potion at mid index is a successful pair.
                if ((long) spells[i] * potions[mid] >= successThreshold) {
                    right = mid; // If successful, narrow the search to the left half.
                } else {
                    left = mid + 1; // If not successful, narrow the search to the right half.
                }
            }

            // The number of successful pairs for the current spell is the total number
            // of potions minus the number of potions that did not meet the success threshold.
            successfulPairs[i] = nPotions - left;
        }
        return successfulPairs; // Return the array of successful pair counts.
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to find the number of successful pairs
    vector<int> successfulPairs(vector<int>& spells, vector<int>& potions, long long success) {
        // Sort the potions array for binary search
        sort(potions.begin(), potions.end());

        // ans will hold the answer to the problem
        vector<int> ans;

        // m is the size of the potions array
        int m = potions.size();

        // Iterate over each spell
        for (int spell : spells) {
            // Find the first potion that, when multiplied with the spell,
            // is greater than or equal to the success threshold.
            // We use 1.0 to cast success to double for correct division.
            int index = lower_bound(potions.begin(), potions.end(), (success + spell - 1) / spell) - potions.begin();

            // Push the number of successful pairs into the answer vector.
            // Subtract the found index from the size to determine the number of successful potions.
            ans.push_back(m - index);
        }

        // Return the final answer
        return ans;
    }
};

```

Typescript Solution

```typescript
function successfulPairs(spells: number[], potions: number[], successThreshold: number): number[] {
    // Sort the potions array in ascending order
    potions.sort((a, b) => a - b);

    // The number of potions available
    const potionCount = potions.length;

    // Initialize an array to store the number of successful pairs for each spell
    const successfulPairsCount: number[] = [];

    // Iterate over each spell
    for (const spell of spells) {
        // Set the initial bounds for binary search
        let left = 0;
        let right = potionCount;

        // Perform binary search to find the index where
        // spell * potionValue is greater than or equal to success threshold
        while (left < right) {
            // Calculate the middle index
            const mid = Math.floor((left + right) / 2);

            // Check if the current combination meets the success requirement
            if (spell * potions[mid] >= successThreshold) {
                // If it does, we need to find if there's a smaller index that also satisfies the condition
                right = mid;
            } else {
                // If not, move the left pointer to narrow the search
                left = mid + 1;
            }
        }

        // The number of potions that meet the success criteria with the current spell
        // is the length of the potions array minus the left bound found by binary search
        successfulPairsCount.push(potionCount - left);
    }

    // Return the array containing the counts of successful spell-potion pairs
    return successfulPairsCount;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code snippet consists of two parts: sorting the `potions` list and performing binary searches using the `bisect_left` function.

1. **Sorting:** The `potions.sort()` operation has a time complexity of `O(n log n)` where `n` is the number of potions.
2. **Binary Search:** The binary search inside the list comprehension using `bisect_left` is performed for each spell. If there are `s` spells, and the binary search has a time complexity of `O(log n)` for each spell, the entire list comprehension has a time complexity of `O(s log n)`.

Therefore, combining both operations, the total time complexity is `O(n log n + s log n)`.

Space Complexity

The space complexity is determined by the additional space required beyond the input data.

1. **Sorting Space:** The sorting is done in place, which does not require additional space, so it's `O(1)`.
2. **Output List:** There is a list comprehension that generates a list of the same length as the number of spells. Therefore, it has a space complexity of `O(s)` where `s` is the number of spells.

Given that `O(s)` is the larger term between `O(1)` and `O(s)`, the overall space complexity is `O(s)`.