1385. Find the Distance Value Between Two Arrays

Easy Array Two Pointers Binary Search Sorting Leetcode Link

Problem Description

The goal of this problem is to compare two lists of integers, `arr1` and `arr2`, and determine the number of elements in `arr1` that are at least distance `d` apart from every element in `arr2`. We define this "distance value" as the count of such `arr1[i]` elements for which no `arr2[j]` exists where `|arr1[i]-arr2[j]| <= d`. In other words, we're counting how many elements in `arr1` are not within `d` distance of any element in `arr2`.

Intuition

The key to solving this problem efficiently is to recognize that we can exploit the properties of sorted arrays. By sorting `arr2`, we can quickly determine if any given element in `arr1` is within the distance `d` to elements in `arr2` using binary search, rather than comparing it to every element in `arr2`.

The intuition behind the solution is to sort `arr2` first. Then, for each element `a` in `arr1`, we perform a binary search to find the position where `a - d` would fit in `arr2`. This tells us the index of the first element in `arr2` that could potentially be within distance `d` of `a`. We call the helper function `check` with the current element `a` to verify that indeed there is no `arr2[j]` such that `|a - arr2[j]| <= d`.

Since `arr2` is sorted, if the element at the found index is greater than `a + d`, then `a` is not within distance `d` of any element in `arr2`. If the found index is equal to the length of `arr2`, it means `a - d` is larger than any element in `arr2`, and thus also satisfies the condition.

Finally, we count and sum up all such `a` from `arr1` that satisfy the condition by using the `sum` function on the generator expression that iterates through `arr1` and applies the `check` function to each element.

Solution Approach

The solution leverages binary search and sorting to efficiently determine if any elements in `arr1` are within `d` distance of elements in `arr2`.

Here is a step-by-step explanation of the algorithm:

1. **Sorting `arr2`:** The first step is to sort the array `arr2`. Sorting is important because it enables us to use binary search, which significantly reduces the time complexity of searching within `arr2` from `O(n)` to `O(log n)` for each element in `arr1`.

2. **Defining a helper function `check`:** Inside the `Solution` class, a helper function named `check` is defined which takes an integer `a` as input. The purpose of this function is to determine if there is an element in `arr2` within `d` distance of `a`.

3. **Performing binary search with `bisect_left`:** This function uses `bisect_left` from Python's `bisect` module. Given a sorted array and a target value, `bisect_left` returns the index where the target should be inserted to keep the array sorted. If we search for `a - d` in `arr2`, this will give us the lowest index `i` at which `a - d` could be inserted without violating the sorting order. This index `i` helps us quickly find the potential candidates in `arr2` that could be within `d` distance from `a`.

4. **Checking the condition:** The helper function then checks if the index `i` is at the end of `arr2` (meaning that `a` is greater than all elements in `arr2` plus `d`), or if the element at index `i` in `arr2` is greater than `a + d`. If either of these conditions is true, it means that none of the elements in `arr2` are within `d` distance of `a`. In this case, the function returns `True`.

5. **Counting elements with the `sum` function:** Finally, in the `findTheDistanceValue` method, the sum function iterates over each element `a` in `arr1` and applies the `check` function to it. This produces a generator of `True` or `False` values, where `True` indicates that `a` is at a valid distance from all elements in `arr2`. The sum function then adds up all the `True` values, which correspond to `1`s, thus counting the elements in `arr1` that satisfy the condition.

By utilizing a sorted array and binary search, the solution effectively reduces the number of comparisons that need to be made, resulting in an efficient algorithm with a lower time complexity suitable for large inputs.

Example Walkthrough

Let us illustrate the solution approach with a small example:

Suppose we have `arr1 = [1, 4, 5, 7]`, `arr2 = [3, 6, 10]`, and `d = 2`.

1. **Sorting `arr2`:** First, we sort `arr2`, but it's already sorted in this case: `arr2 = [3, 6, 10]`.

2. **Defining a helper function `check`:** The `check` function will determine if `arr1[i]` is at least distance `d` apart from every element in `arr2`.

3. **Performing binary search with `bisect_left`:**

    - For `a = 1` from `arr1`: We find the place where `1 - 2` (which is `-1`) would fit in `arr2` using `bisect_left`. The index returned is `0` because `-1` is less than the first element in `arr2`. Since `arr2[0]` is `3`, and `3` is greater than `1 + 2` (`3` is not within the distance `d` from `1`), the condition is satisfied.
    - For `a = 4` from `arr1`: `bisect_left` of `4 - 2` will return index `1` because `2` fits between `3` and `6`. However, the element at index `1` is `6`, which is not greater than `4 + 2`; therefore, the condition is not satisfied, and `4` is not at the required distance from an element in `arr2`.
    - For `a = 5` from `arr1`: Using the same approach, the binary search returns index `2` (for `5 - 2`), and `arr2[2]` is `10`, which is greater than `5 + 2`; hence, `5` is at a valid distance from all elements in `arr2`.
    - For `a = 7` from `arr1`: The binary search for `7 - 2` returns index `2` as well, and since `10` is still greater than `7 + 2`, `7` also satisfies the condition.

4. **Checking the condition:** As explained above, after performing the binary search, the helper function checks the conditions to confirm if `a` is at the required distance from all elements in `arr2`.

5. **Counting elements with the `sum` function:** We have found that elements `1`, `5`, and `7` from `arr1` satisfy the condition of being at least distance `d` from all elements of `arr2`. Hence, our distance value would be the count of these elements, which is `3`.

Following the approach above, the `findTheDistanceValue` method would return `3` for the given example, as there are three elements in `arr1` that are at least distance `d` away from every element in `arr2`.

Python Solution

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

class Solution:
    def findTheDistanceValue(self, arr1: List[int], arr2: List[int], d: int) -> int:
        # Helper function to check if there's an element in arr2 within
        # the distance d of the element 'element_from_arr1'.
        def is_valid(element_from_arr1: int) -> bool:
            # Find the position in arr2 where 'element_from_arr1' - d could be inserted
            # to maintain the sorted order.
            index = bisect_left(arr2, element_from_arr1 - d)
            # Return True if either:
            # - index is at the end of arr2 (no element within distance d), or
            # - the element at the found index is greater than 'element_from_arr1' + d
            return index == len(arr2) or arr2[index] > element_from_arr1 + d

        # Sort the second array to leverage binary searching.
        arr2.sort()

        # Use list comprehension to iterate over arr1, applying 'is_valid'
        # function to each element, and sum the results. The result will be
        # the count of elements in arr1 that satisfy the distance value condition.
        return sum(is_valid(element) for element in arr1)

```

Java Solution

```java
class Solution {

    // This function finds the distance value between two arrays.
    public int findTheDistanceValue(int[] arr1, int[] arr2, int d) {
        // Sort the second array to perform binary search later.
        Arrays.sort(arr2);

        // Initialize answer to count the number of elements meeting the condition.
        int answer = 0;

        // Loop over each element in the first array.
        for (int elemArr1 : arr1) {
            // Check if the element in the first array meets the distance condition.
            if (isDistanceMoreThanD(arr2, elemArr1, d)) {
                // Increment the answer if the condition is met.
                answer++;
            }
        }
        // Return the number of elements that meet the condition.
        return answer;
    }

    // Helper function to check if the distance between an element and all elements in another array is more than d.
    private boolean isDistanceMoreThanD(int[] sortedArr2, int elemArr1, int d) {
        int left = 0;
        int right = sortedArr2.length;

        // Perform a binary search to find if there exists an element in arr2 that is within distance d of elemArr1.
        while (left < right) {
            int mid = left + (right - left) / 2; // Avoid potential overflow compared to (left + right) / 2.
            if (sortedArr2[mid] >= elemArr1 - d) {
                // If the middle element is within the lower bound of the distance, narrow the search to the left part.
                right = mid;
            } else {
                // Otherwise, narrow the search to the right part.
                left = mid + 1;
            }
        }

        // After the binary search, if the left index is out of bounds, it means all elements in arr2 are less than elemArr1 - d.
        // If the left index points to an element, that element must be greater than elemArr1 + d to satisfy the condition.
        return left >= sortedArr2.length || sortedArr2[left] > elemArr1 + d;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm>

class Solution {
public:
    // Function to calculate the distance value between two arrays
    int findTheDistanceValue(std::vector<int>& arr1, std::vector<int>& arr2, int d) {
        // Lambda function to check if no element in arr2 lies within the range [a - d, a + d]
        // If true, a contributes to the distance value
        auto isDistanceValid = [&](int a) -> bool {
            // Finding the first element in arr2 which is not less than a - d
            auto it = std::lower_bound(arr2.begin(), arr2.end(), a - d);
            // Check if the iterator reached the end (no such element) or the found element is greater than a + d
            return it == arr2.end() || *it > a + d;
        };

        // Sort array arr2 to use binary search (required by std::lower_bound)
        std::sort(arr2.begin(), arr2.end());

        // Initialize the distance value answer
        int ans = 0;

        // Iterate over each element in arr1
        for (int a : arr1) {
            // Increment the distance value answer if the element a satisfies the isDistanceValid condition
            ans += isDistanceValid(a);
        }

        // Return the computed distance value
        return ans;
    }
};

```

Typescript Solution

```typescript
function findTheDistanceValue(arr1: number[], arr2: number[], d: number): number {
    // Helper function that checks if any element in arr2 is within d distance of element 'value'
    const isElementDistanceValid = (value: number): boolean => {
        let left = 0;
        let right = arr2.length;
        while (left < right) {
            // Find the middle index
            const middle = Math.floor((left + right) / 2);
            // Check if the middle element is within the allowed distance
            if (arr2[middle] >= value - d) {
                right = middle; // Element is too close, adjust the search range to the left
            } else {
                left = middle + 1; // Element is not close enough, adjust the search range to the right
            }
        }
        // If left is equal to length of arr2 or the element at 'left' index is outside the distance 'd' from 'value', return true
        return left === arr2.length || arr2[left] > value + d;
    };

    // Sort the second array to enable binary search
    arr2.sort((a, b) => a - b);
    // Initialize the count of elements that satisfy the condition
    let validElementCount = 0;
    // Iterate through the elements of arr1
    for (const value of arr1) {
        // Check if the current element satisfies the distance condition for every element in arr2
        if (isElementDistanceValid(value)) {
            // If condition is met, increment the count
            validElementCount++;
        }
    }
    // Return the final count of valid elements
    return validElementCount;
}

```

Time and Space Complexity

The given Python code defines a function `findTheDistanceValue` that computes the distance value between two lists `arr1` and `arr2`, given a distance `d`. Here's an analysis of the time and space complexity:

Time Complexity:

1. Sorting `arr2` using `arr2.sort()`: The sort operation has a time complexity of `O(n log n)` where `n` is the length of `arr2`.
2. Iterating through `arr1` with the for loop: The loop runs `m` times where `m` is the length of `arr1`.
3. The `check` function calls `bisect_left`, a binary search function, for every element `a` in `arr1`. The binary search runs in `O(log n)` time.
4. Multiplying the above two factors, the for loop with the binary search operation results in a time complexity of `O(m log n)`.

Adding both parts, the overall time complexity of the code is dominated by the `O(m log n)` part (since this part depends both on the length of `arr1` and the fact that a binary search is performed on `arr2`), assuming `m log n` > `n log n`, which might be the case if `m` is significantly larger than `n` or vice versa. Therefore, the total time complexity is `O(m log n)`.

Space Complexity:

1. The sorted version of `arr2`: Python's `sort()` function sorts the list in place, so it doesn't use any additional space other than a small constant amount, hence it has a space complexity of `O(1)`.
2. The check function itself and the binary search do not use extra space that scales with the input size (they use a constant amount of extra space).

Therefore, the space complexity of the entire function is `O(1)` as there is no additional space used that depends on the input size of `arr1` and `arr2`.