

# 1855. Maximum Distance Between a Pair of Values

Medium Greedy Array Two Pointers Binary Search

## Problem Description

In this problem, we are given two non-increasing integer arrays, `nums1` and `nums2`. We define a valid pair as a pair of indices  $(i, j)$  such that:

- `i` is an index from `nums1` and `j` is an index from `nums2`.
- `i` must be less than or equal to `j` ( $i \leq j$ ).
- The value at `nums1[i]` must be less than the value at `nums2[j]` ( $nums1[i] < nums2[j]$ ).

The distance for a valid pair  $(i, j)$  is calculated by the difference  $j - i$ . The goal of the problem is to return the maximum possible distance for any valid pair of indices. If there are no valid pairs, we should return `0`.

Note that an array is non-increasing if each element is less than or equal to the previous element. This implies that the numbers in both `nums1` and `nums2` are sorted in non-increasing order (from largest to smallest).

## Intuition

The intuition behind the solution involves recognizing that if a pair  $(i, j)$  is valid, then any pair  $(i, k)$  where  $k < j$  is also valid due to the non-increasing order of the arrays. This property allows us to use a two-pointer or a [binary search](#) approach to efficiently find the maximum distance. We choose the two-pointer approach here as it simplifies the implementation.

We start by initializing [two pointers](#), `i` and `j`, at the beginning of `nums1` and `nums2` respectively. We then iterate through `nums1` with `i`, trying to find the furthest `j` in `nums2` that will form a valid pair. As we go, we keep track of the maximum distance `ans`.

The crucial observation is that since both `nums1` and `nums2` are non-increasing, once we find a `j` for a particular `i` such that `nums1[i] <= nums2[j]`, we don't need to reset `j` back to `i` for the next `i`. Instead, we can continue from the current `j`, because if `nums1[i] <= nums2[j]` and `nums1` is non-increasing, then `nums1[i + 1]` will be less than or equal to `nums1[i]`, and hence also less than or equal to `nums2[j]`. We keep incrementing `j` until the condition `nums1[i] <= nums2[j]` is no longer met.

However, we need to calculate the distance  $j - i$  and store the maximum. Since `j` is incremented in the inner loop until `nums1[i] <= nums2[j]` is false, at this point `j` is actually one index past the valid pair, so we compute  $j - i - 1$  to get the correct distance for considering the current index `i`.

As a result, the algorithm efficiently calculates the maximum distance of any valid pair by scanning through both arrays only once.

## Solution Approach

The given reference solution approach indicates using a [Binary Search](#) algorithm. However, the provided solution code actually uses a two-pointer technique, which intuitively exploits the sorted nature of the arrays to find the maximum distance. Below, we'll walk through the implementation of the two-pointer solution given in the code.

The code defines a class `Solution` with a method `maxDistance` which accepts two arrays, `nums1` and `nums2`.

- Initialization:** We begin by initializing a few variables:
  - `m`, the length of `nums1`
  - `n`, the length of `nums2`
  - `ans`, which will keep track of the maximum distance found among valid pairs. It is set to `0` initially.
  - [Two pointers](#) `i` and `j`, both set to `0`, to iterate through `nums1` and `nums2` respectively.
- Iterating with Two Pointers:** We use a while loop to iterate through `nums1` with `i` as long as  $i < m$  (ensuring we don't go out of bounds).
- Finding the Furthest j:** For every index `i`, we have an inner while loop that seeks the furthest `j` such that `nums1[i] <= nums2[j]`. This loop runs as long as  $j < n$ . If the condition is met, `j` is incremented by `1`, moving the second pointer ahead in `nums2`.
- Tracking the Maximum Distance:** Once we are either out of bounds in `nums2` or the condition is no longer satisfied, the inner loop breaks and we use the maximum function `max(ans, j - i - 1)` to update our answer with the largest distance found so far. The `-1` is there to correct for the `j` index that is now at an invalid pair, having just moved past the last valid `nums2[j]`.
- Incrementing i:** After checking and possibly updating the `ans` with the maximum distance for the current `i`, we increment `i` by `1` and repeat the above steps until all elements of `nums1` have been considered.
- Returning the Answer:** When the loop terminates, `ans` will hold the maximum distance of any valid pair  $(i, j)$ , or it will remain `0` if no valid pairs existed. This value is then returned as the result of the method.

The two-pointer approach is efficient because it leverages the sorted properties of `nums1` and `nums2` to avoid redundant checks. It doesn't require additional data structures and runs in linear time, as each pointer only passes through its respective array once.

Thus, the implementation uses a methodical two-pointer strategy to measure the maximum distance between valid pairs across two sorted arrays.

## Example Walkthrough

Let's walk through an example to illustrate the solution approach in practice.

Suppose we have the following inputs: `nums1 = [4, 2, 2]` `nums2 = [3, 2, 0]`

We want to find the maximum possible distance for any valid pair of indices.

- We start with `i = 0` in `nums1` and `j = 0` in `nums2`, with `ans = 0` as we haven't found any valid pairs yet.
- In the outer loop, `i` is now at index `0` and `nums1[0]` is `4`. The inner while loop starts iterating `j` through `nums2`. `nums2[j]` is `3`, which is less than `nums1[i]`, so we don't have a valid pair. We continue to the next `j` without changing `ans`.
- Now, `j = 1` and `nums2[j]` is `2`. It's still not greater than `nums1[i] = 4`, so we move to the next `j`.
- At `j = 2`, we have `nums2[j]` equal to `0`, which is also not greater than `4`. Having reached the end of `nums2` without finding a valid pair for `i = 0`, we increment `i` to `1` and start over with `j = 0`.
- Now `i = 1` and `nums1[i]` is `2`. Starting again from the beginning of `nums2`, we find that `nums2[0]` is `3`, which is greater than `nums1[1]`. This is a valid pair (`i=1, j=0`). Hence, we update `ans = max(ans, 0 - 1 - 1)`, but this gives us `-2`, which is not an increase over `0`, so `ans` stays at `0`.
- We increment `j` to see if there is a pair with a larger distance. Now `j` is `1` and since `nums1[i] = 2` is already less than `nums2[j] = 2`, we stop the inner while loop and update `ans`. Now `ans = max(0, 1 - 1 - 1)`, which remains `0` as the indices are the same.
- Finally, we increment `i` to `2`. `nums1[i]` is still `2`, and we try with `nums2[j]` starting from the beginning. At `j = 0`, we once again have a valid pair (`i=2, j=0`) and update `ans` to `max(0, 0 - 2 - 1)`, giving us `-3`, which doesn't increase `ans`. Since all values are non-increasing and there's no greater element in `nums2` than `3`, we can conclude there's no need to iterate `j` further for `i = 2`.

After considering all elements of `nums1` and finding no valid pair with a positive distance, the function returns the `ans`, which in this case remains `0` as there were no valid pairs that satisfied the condition for a positive maximum distance.

This example demonstrates how the two-pointer approach navigates through both arrays, cleverly using the condition of non-increasing order to find the valid pairs and calculating the maximum possible distance.

## Solution Implementation

### Python

```
# Import the List type from the typing module for type annotations
from typing import List

class Solution:
    def maxDistance(self, nums1: List[int], nums2: List[int]) -> int:
        # Initialize the lengths of the two input lists
        len_nums1, len_nums2 = len(nums1), len(nums2)

        # Initialize the maximum distance and the indices for nums1 and nums2
        max_distance = idx_nums1 = idx_nums2 = 0

        # Loop through the elements of nums1
        while idx_nums1 < len_nums1:
            # For the current idx nums1, increase idx nums2 as long as
            # the conditions are satisfied (nums1[idx_nums1] <= nums2[idx_nums2])
            while idx_nums2 < len_nums2 and nums1[idx_nums1] <= nums2[idx_nums2]:
                idx_nums2 += 1

            # Update the maximum distance if a larger one is found
            # Decrement by 1 because idx nums2 is increased one more time before the condition fails
            max_distance = max(max_distance, idx_nums2 - idx_nums1 - 1)
            # Move to the next index in nums1
            idx_nums1 += 1

        # Return the maximum distance found
        return max_distance
```

### Java

```
class Solution {
    public int maxDistance(int[] nums1, int[] nums2) {
        int lengthNums1 = nums1.length; // length of the first array
        int lengthNums2 = nums2.length; // length of the second array
        int maxDist = 0; // variable to keep track of the maximum distance

        // Initialize two pointers for both arrays
        for (int indexNums1 = 0, indexNums2 = 0; indexNums1 < lengthNums1; ++indexNums1) {
            // Move the indexNums2 pointer forward as long as the condition holds
            while (indexNums2 < lengthNums2 && nums1[indexNums1] <= nums2[indexNums2]) {
                ++indexNums2;
            }
            // Update maxDist with the maximum distance found so far
            // We subtract 1 because indexNums2 has moved one step further than the true distance
            maxDist = Math.max(maxDist, indexNums2 - indexNums1 - 1);
        }
        return maxDist; // return the maximum distance found
    }
}
```

### C++

```
class Solution {
public:
    int maxDistance(vector<int>& nums1, vector<int>& nums2) {
        int nums1Size = nums1.size(); // Represents the size of the first vector nums1
        int nums2Size = nums2.size(); // Represents the size of the second vector nums2
        int maxDist = 0; // Initialize maximum distance to 0

        // Using two pointers, iterate through both arrays
        for (int i = 0, j = 0; i < nums1Size; ++i) {
            // While i is within the bounds of nums2 and the value at nums1[i]
            // is less than or equal to the value at nums2[j], increment j
            while (i < nums2Size && nums1[i] <= nums2[j]) {
                ++j;
            }
            // Calculate the current distance, and update maxDist if it's greater
            // than the current maxDist. Subtract one because i has been incremented
            // in the last iteration of the inner loop where the condition was still valid.
            maxDist = max(maxDist, j - i - 1);
        }

        return maxDist; // Return the computed maximum distance
    }
};
```

### TypeScript

```
function maxDistance(nums1: number[], nums2: number[]): number {
    let maximumDistance = 0; // This will store the maximum distance found.
    const nums1Length = nums1.length; // Length of the first array
    const nums2Length = nums2.length; // Length of the second array

    // Start with two pointers, 'i' for array nums1 and 'j' for array nums2
    for (let i = 0, j = 0; i < nums1Length; ++i) {
        // Increment 'j' as long as it's within the bounds of nums2
        // and the element in nums1 is less than or equal to the element in nums2.
        // This ensures that the 'j' finds the furthest distance it can go for each 'i'.
        while (i < nums2Length && nums1[i] <= nums2[j]) {
            j++;
        }

        // Update the maximum distance. Since 'j' moved one step extra, we subtract 1.
        // i.e., j - i - 1 is the total distance from current 'i' to the last valid 'j'
        // but we need to subtract 1 since 'j' has moved one step ahead of valid position.
        maximumDistance = Math.max(maximumDistance, j - i - 1);
    }

    // Return the maximum distance found
    return maximumDistance;
}
```

```
# Import the List type from the typing module for type annotations
from typing import List

class Solution:
    def maxDistance(self, nums1: List[int], nums2: List[int]) -> int:
        # Initialize the lengths of the two input lists
        len_nums1, len_nums2 = len(nums1), len(nums2)

        # Initialize the maximum distance and the indices for nums1 and nums2
        max_distance = idx_nums1 = idx_nums2 = 0

        # Loop through the elements of nums1
        while idx_nums1 < len_nums1:
            # For the current idx nums1, increase idx nums2 as long as
            # the conditions are satisfied (nums1[idx_nums1] <= nums2[idx_nums2])
            while idx_nums2 < len_nums2 and nums1[idx_nums1] <= nums2[idx_nums2]:
                idx_nums2 += 1

            # Update the maximum distance if a larger one is found
            # Decrement by 1 because idx nums2 is increased one more time before the condition fails
            max_distance = max(max_distance, idx_nums2 - idx_nums1 - 1)
            # Move to the next index in nums1
            idx_nums1 += 1

        # Return the maximum distance found
        return max_distance
```

## Time and Space Complexity

### Time Complexity

The time complexity of the given code is  $O(m + n)$ . Here, `m` is the length of `nums1` and `n` is the length of `nums2`. This is because the two pointers `i` and `j` only move forward through their respective arrays, with `i` incrementing in every outer loop, and `j` potentially moving forward during each inner loop until `nums1[i] <= nums2[j]` no longer holds. Since each pointer goes through its respective array at most once, the time complexity is linear with respect to the sizes of the input arrays.

### Space Complexity

The space complexity of the code is  $O(1)$ . No additional space is used that grows with the input size. The variables `ans`, `i`, `j`, `m`, and `n` each use a constant amount of space.