624. Maximum Distance in Arrays

Medium Greedy Array

Leetcode Link

Problem Description

In this problem, we are provided `n` separate arrays, each being sorted in ascending order. Our task is to find the maximum distance between any two integers where each integer is taken from a different array. The distance is defined as the absolute difference between those two integers. That is, if we take one integer `a` from one array and another integer `b` from a different array, the distance is `|a - b|`. We need to compute and return the maximum such distance possible.

Intuition

To solve this problem efficiently, we leverage the fact that each of the `n` arrays is sorted in ascending order. Given this property, the smallest element of each array will be at the 0-th index and the largest element at the last index.

To find the maximum distance, we need to make the difference as large as possible. This means we should consider the potential largest distance by taking the smallest element from one array and the largest from another array. So to maximize the distance, we always consider the minimum element from one array and the maximum element from the other.

The approach is to iterate through the arrays while tracking the smallest and largest elements we have seen so far. For each new array, we consider the distance between the current array's smallest element and the maximum element seen so far, and vice versa - the distance between the current array's largest element and the smallest element seen so far. The answer is the largest of all these distances.

Here is the step-by-step reasoning:

1. Initialize `ans` to 0, which will keep track of the maximum distance.
2. Take the smallest (`mi`) and largest (`mx`) elements from the first array to initiate tracking.
3. Iterate through the remaining arrays starting from the second one. For each array:
    - Calculate the absolute difference between the current array's smallest element and `mx` (the largest element seen so far).
    - Calculate the absolute difference between the current array's largest element and `mi` (the smallest element seen so far).
    - Update `ans` with the largest value between `ans`, the first calculated difference, and the second calculated difference.
    - Update `mi` with the smallest value between `mi` (the smallest value found so far) and the smallest element of the current array.
    - Update `mx` with the largest value between `mx` (the largest value found so far) and the largest element of the current array.
4. After finishing the iteration, return `ans` as the maximum distance found.

By only comparing the extreme ends of each array, we ensure that we are always considering the likely pairs to give us the maximum distance while maintaining a linear time complexity.

Solution Approach

The solution follows a greedy strategy by keeping track of the smallest and the largest element found so far across all arrays. Here is how the implementation reflects the approach:

- An initial answer `ans` is set to 0, which will hold the maximum distance.
- The smallest element `mi` and the largest element `mx` from the first array are used to initialize the tracking.
- Then, for every subsequent array, the implementation performs the following steps:
    1. Calculates two distances: the distance between the smallest element of the current array and the largest element `mx` seen so far (`a = abs(arr[0] - mx)`), and the distance between the largest element of the current array and the smallest element `mi` seen so far (`b = abs(arr[-1] - mi)`).
    2. The answer `ans` is updated to be the maximum of `ans`, `a`, and `b`.
    3. The current array's smallest element is considered to update `mi` using `mi = min(mi, arr[0])` if it is smaller than the current `mi`.
    4. Similarly, the current array's largest element is used to update `mx` using `mx = max(mx, arr[-1])` if it is larger than the current `mx`.

By using `min()` and `max()` functions, the elements `mi` and `mx` are continuously updated to always represent the smallest and largest element found up to the current point in iteration.

This approach guarantees each array is only visited once, making the overall time complexity `O(n)`, where `n` is the number of arrays. There is no use of additional data structures, keeping space complexity to `O(1)` as it only uses a few variables for tracking the minimum, maximum, and the current answer.

The code directly follows this strategy and does not use any complex data structures or algorithms beyond basic conditional statements and math operations.

Here's a snippet of the main logic using the algorithm described:

```
class Solution:
    def maxDistance(self, arrays: List[List[int]]) -> int:
        ans = 0
        mi, mx = arrays[0][0], arrays[0][-1]
        for arr in arrays[1:]:
            a, b = abs(arr[0] - mx), abs(arr[-1] - mi)
            ans = max(ans, a, b)
            mi = min(mi, arr[0])
            mx = max(mx, arr[-1])
        return ans
```

Example Walkthrough

Let's say we have the following 3 sorted arrays:

- **Array 1**: `[1, 2, 3]`
- **Array 2**: `[4, 5]`
- **Array 3**: `[1, 5, 9]`

Our goal is to find the maximum distance between any two integers from two different arrays.

1. Initialize `ans` to 0, which will be used to keep track of the maximum distance found so far.
2. We start with the first array and take its smallest and largest elements as the initial values for `mi` and `mx` respectively. In this case, `mi` is `1` and `mx` is `3`.

Now, we iterate through the remaining arrays to calculate potential distances and update `mi` and `mx`.

3. Moving to the second array `[4, 5]`, we calculate the distance between the smallest element in this array `4` and the current `mx` `3` which is `abs(4 - 3) = 1`. We also calculate the distance between the largest element in this array `5` and the current `mi` `1` which is `abs(5 - 1) = 4`. We update `ans` to the maximum of `ans`, which is `0`, and these new distances. `ans` is now `4`. We also update `mi` and `mx` if necessary. Since `4` is greater than `mi` and `5` is greater than `mx`, we only update `mx` to `5`.
4. Next, we go to the third array `[1, 5, 9]`. We calculate the distance between `1` (the smallest element of this array) and the current largest `mx`, which is `5`. The distance is `abs(1 - 5) = 4`. Then, we calculate the distance between `9` (the largest element of this array) and the current smallest `mi`, which is `1`. The distance here is `abs(9 - 1) = 8`. We update `ans` to the maximum of `ans` and these new distances. `ans` is now updated to `8`, which is the maximum distance found. We check if we need to update `mi` or `mx`. Since `1` is equal to current `mi` and `9` is greater than current `mx`, we update `mx` to `9`.

Finally, after iterating through all arrays, we have the largest distance `ans` as `8`, which is the correct maximum distance between two integers from two different arrays. Thus, we return `8`.

Python Solution

```
from typing import List

class Solution:
    def maxDistance(self, arrays: List[List[int]]) -> int:
        # Initialize the maximum distance and the minimum and maximum values
        max_distance = 0
        min_value, max_value = arrays[0][0], arrays[0][-1]

        # Iterate over the arrays starting from the second one
        for array in arrays[1:]:
            # Compute the potential distances between the current array's first or last element
            # with the opposite ends of the known range (max_value and min_value)
            distance_min_to_max = abs(array[0] - max_value)
            distance_max_to_min = abs(array[-1] - min_value)

            # Update max_distance to be the maximum of the current max_distance and the new distances
            max_distance = max(max_distance, distance_min_to_max, distance_max_to_min)

            # Update the overall min_value and max_value with the current array's values
            min_value = min(min_value, array[0])
            max_value = max(max_value, array[-1])

        # Return the computed maximum distance between any two pairs
        return max_distance
```

Java Solution

```
class Solution {

    // Method to find the maximum distance between any pair of elements
    // from different arrays within the list of arrays.
    public int maxDistance(List<List<Integer>> arrays) {
        // Initializing 'ans' as 0 to hold the maximum distance encountered.
        int maxDistance = 0;

        // Assign the first element of the first array as the minimum known value 'min'.
        int min = arrays.get(0).get(0);

        // Assign the last element of the first array as the maximum known value 'max'.
        int max = arrays.get(0).get(arrays.get(0).size() - 1);

        // Iterate over the remaining arrays in the list starting from the second array.
        for (int i = 1; i < arrays.size(); ++i) {
            List<Integer> array = arrays.get(i);

            // Calculate the absolute difference between the first element of the current array
            // and the known maximum. This represents a potential max distance.
            int distanceWithMax = Math.abs(array.get(0) - max);

            // Calculate the absolute difference between the last element of the current array
            // and the known minimum. This represents a potential max distance.
            int distanceWithMin = Math.abs(array.get(array.size() - 1) - min);

            // Update 'maxDistance' with the greater of the two newly calculated distances
            // if it either is larger than the current 'maxDistance'.
            maxDistance = Math.max(maxDistance, Math.max(distanceWithMax, distanceWithMin));

            // Update the known minimum value 'min' if the first item of the current array is smaller.
            min = Math.min(min, array.get(0));

            // Update the known maximum value 'max' if the last item of the current array is larger.
            max = Math.max(max, array.get(array.size() - 1));
        }

        // Return the largest distance found.
        return maxDistance;
    }
}
```

C++ Solution

```
#include <vector>
#include <algorithm> // include the algorithm header for using min, max functions

class Solution {
public:
    int maxDistance(vector<vector<int>>& arrays) {
        int maxDistance = 0; // maximum distance found so far
        int minElement = arrays[0][0]; // initialize with the first element of the first array
        int maxElement = arrays[0].back(); // initialize with the last element of the first array

        // Start from the second array
        for (int i = 1; i < arrays.size(); ++i) {
            // Reference to the current array to avoid copying
            auto& currentArray = arrays[i];

            // Calculate distance between the smallest element so far and the last element of the current array
            int distanceToMin = abs(currentArray[0] - maxElement);

            // Calculate distance between the largest element so far and the first element of the current array
            int distanceToMax = abs(currentArray.back() - minElement);

            // Update maxDistance with the largest of the three: itself, distanceToMin, distanceToMax
            maxDistance = max({maxDistance, distanceToMin, distanceToMax});

            // Update minElement with the smaller between minElement and the first element of the current array
            minElement = min(minElement, currentArray[0]);

            // Update maxElement with the larger between maxElement and the last element of the current array
            maxElement = max(maxElement, currentArray.back());
        }

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

Typescript Solution

```
// Importing the first and last functions from lodash for array manipulation
import { first, last } from 'lodash';

// Function to calculate the maximum distance between any pair of elements from different arrays
function maxDistance(arrays: number[][]): number {
    // Maximum distance found so far
    let maxDistance = 0;
    // Initialize with the first element of the first sub-array
    let minElement = first(arrays[0]) as number;
    // Initialize with the last element of the first sub-array
    let maxElement = last(arrays[0]) as number;

    // Start from the second sub-array
    for (let i = 1; i < arrays.length; ++i) {
        // Reference to the current sub-array to avoid copying
        const currentArray = arrays[i];

        // Calculate distance between the smallest element so far and the last element of the current sub-array
        const distanceToMin = Math.abs(first(currentArray) as number - maxElement);

        // Calculate distance between the largest element so far and the first element of the current sub-array
        const distanceToMax = Math.abs(last(currentArray) as number - minElement);

        // Update maxDistance with the largest of the three: itself, distanceToMin, distanceToMax
        maxDistance = Math.max(maxDistance, distanceToMin, distanceToMax);

        // Update minElement with the smaller between minElement and the first element of the current sub-array
        minElement = Math.min(minElement, first(currentArray) as number);

        // Update maxElement with the larger between maxElement and the last element of the current sub-array
        maxElement = Math.max(maxElement, last(currentArray) as number);
    }

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

Time and Space Complexity

Time Complexity

The time complexity of the given code is primarily determined by the loop that iterates over all elements of the `arrays` list, except the first element. In the worst case, where there are `n` arrays inside the `arrays` list, we iterate `n - 1` times. Inside the loop, we execute a constant number of operations for each array: calculating the absolute difference between the first and last elements with `mi` and `mx`, updating `ans`, and updating `mi` and `mx` with the current array's first and last elements, respectively. Since all these operations inside the loop have a constant time complexity, the overall time complexity of the loop is `O(n)`.

Hence, the total time complexity of the code is `O(n)`.

Space Complexity

The space complexity is determined by the additional memory used by the program which is not part of the input. In the given code, we use a fixed number of variables (`ans`, `mi`, `mx`, `a`, `b`) that do not depend on the size of the input. No other dynamic data structures or recursive calls which could use additional space are involved. Therefore, the space complexity is `O(1)`, which represents constant space usage.