1537. Get the Maximum Score

Two Pointers

Dynamic Programming

Problem Description

Greedy Array

Hard

You are presented with two distinct sorted arrays nums1 and nums2. Your task is to find the maximum score you can achieve by traversing these arrays according to a set of rules. A valid path through the arrays is defined by starting at the first element of either nums1 or nums2 and then moving from left to right. If, during your traversal, you encounter a value that exists in both nums1 and nums2, you are allowed to switch to the other array—but only once for each value. The score is the sum of all unique values that you encounter on this path. The goal is to find the maximum possible score for all valid paths. As the final score might be large, you should return the result modulo 10^9 + 7.

ntuition

To solve this problem, one needs to understand that the maximum score is obtained by traversing through the maximum values from both arrays. Since you are allowed to switch between arrays at common elements, you should do so in a way that maximizes

the score.

The key is to walk through both arrays in parallel and track two sums: one for nums1 and another for nums2. As long as the numbers in both arrays are distinct, keep adding them to their respective sums. When a common element is met, you have a choice to switch paths. You want to take the path with the greater sum up to this point since that gives you the maximum score.

From that common element, reset both sums to the greater sum and continue the process. By doing this, you effectively collect the maximum values from both arrays and switch paths at the optimal times to ensure you are always on the path with the greatest potential score. When you reach the end of both arrays, the maximum sum at that point will be the maximum score that can be obtained.

Solution Approach The solution is an elegant application of the two-pointer technique, which is well-suited for problems involving sorted arrays.

Because the two arrays are sorted and the problem description allows us to switch paths at common elements, the two-pointer

method allows us to efficiently compare elements from both arrays without needing additional space. Here's how the given

solution works step by step:

to switch paths.

Use a while loop to continue processing until both pointers have reached the end of their respective arrays (i < m or j < n). Inside the loop, there are several cases to consider:

Initialize two sums, f and g, to keep track of the running sum for nums1 and nums2. These variables will also help decide when

a. If pointer i has reached the end of nums1, add the current element in nums2 to sum g, and increment j.

Initialize two pointers, i and j, to start at the beginning of nums1 and nums2, respectively.

- b. If pointer j has reached the end of nums2, add the current element in nums1 to sum f, and increment i.
- c. If the current element in nums1 (nums1[i]) is less than the current element in nums2 (nums2[j]), add nums1[i] to sum f and increment i.

d. If the current element in nums1 is greater than the current element in nums2, add nums2[j] to sum g and increment j.

e. If the current elements in both arrays are equal, a common element is encountered. The max of f and g (which represents

you could end in either array.

potential score at each step.

nums1 = [2, 4, 5, 8, 10]

nums2 = [4, 6, 8, 9]

Example Walkthrough

- the best score up to this point from either array) is added to the common element, then this value is assigned to both f and g since switching paths here is allowed and should yield the maximum score. Increment both i and j after this operation.
- score. In terms of data structures, no additional storage is needed apart from a few variables to keep track of the indices and the sums.

This solution's space complexity is O(1), only requiring constant space, and the time complexity is O(m+n), where m and n are the

lengths of the two input arrays. Because each element in the arrays is examined at most once by each pointer, the algorithm is

After the loop ends (both arrays have been fully traversed), the final answer is the maximum of the two sums, f and g, since

Return this maximum value modulo 10^9 + 7 as per the problem statement requirements to account for a potentially large

highly efficient. This solution is a demonstration of cleverly optimizing the process of path selection in a way that continually maximizes the

Following the proposed solution, here's the walkthrough: We initialize two pointers i = 0 for nums 1 and j = 0 for nums 2. We also initialize two sums f = 0 and g = 0.

Now we compare nums1[i] (which is 4) with nums2[j] (also 4). Since they are equal, we have a common element.

b. We add the maximum of f and g to the common value and assign it to both f and g. Thus, f = g = 4 + 2 = 6.

Now, nums1[i] is 5 and nums2[j] is 6. Since 5 < 6, we add nums1[i] to f and increment i. Now, f = 11, g = 6.

Since nums1[0] < nums2[0], we add nums1[i] (which is 2) to f and increment i. Now, f = 2, g = 0.

a. We switch to the array with the higher sum, which are equal at this moment (f = g = 2), so the path doesn't change.

```
c. Increment both i and j. Now, i = 2, j = 1.
```

a. We choose the path with the higher sum, which is f (11 > 6).

Let's take two sorted arrays to illustrate the solution approach:

```
We have nums1[i] as 8 and nums2[j] also 8. Another common element encountered.
```

29 itself since it's much less than the modulo value.

MOD = 10**9 + 7 # The modulo value for the result

sum_nums1 += nums1[index_nums1]

sum_nums1 += nums1[index_nums1]

sum_nums2 += nums2[index_nums2]

Return the maximum of the two sums modulo MOD

int maxSum(std::vector<int>& nums1, std::vector<int>& nums2) {

int size1 = nums1.size(), size2 = nums2.size();

// Iterate through both arrays simultaneously.

} else if (nums1[index1] < nums2[index2]) {</pre>

} else if (nums1[index1] > nums2[index2]) {

// and increment both pointers as the path merges.

// Return the maximum sum path lesser array modulo the defined mod value.

len_nums1, len_nums2 = len(nums1), len(nums2) # Lengths of the input arrays

index_nums1 = index_nums2 = 0 # Index pointers for nums1 and nums2

sum_nums1 = sum_nums2 = 0 # Running sums for nums1 and nums2

Process both arrays until we reach the end of one of them

nums1MaxSum = max0fBoth;

nums2MaxSum = max0fBoth;

return Math.max(nums1MaxSum, nums2MaxSum) % mod;

def maxSum(self, nums1: List[int], nums2: List[int]) -> int:

MOD = 10**9 + 7 # The modulo value for the result

nums1Index++;

nums2Index++;

Initialize variables

const maxOfBoth = Math.max(nums1MaxSum, nums2MaxSum) + nums1[nums1Index];

while (index1 < size1 || index2 < size2) {</pre>

sum2 += nums2[index2++];

sum1 += nums1[index1++];

sum1 += nums1[index1++];

sum2 += nums2[index2++];

} else if (index2 == size2) {

if (index1 == size1) {

} else {

const int MODULO = 1e9 + 7; // Define the modulo constant.

int index1 = 0, index2 = 0; // Initialize pointers for the two arrays.

long long sum1 = 0, sum2 = 0; // Initialize sums to 0 as long long to prevent overflow.

// If nums1 is exhausted, add remaining elements of nums2 to sum2.

// If nums2 is exhausted, add remaining elements of nums1 to sum1.

// If the current element in nums1 is less than in nums2, add it to sum1.

// If the current element in nums2 is less than in nums1, add it to sum2.

index_nums1 += 1

index_nums1 += 1

index_nums2 += 1

index_nums1 += 1

index_nums2 += 1

return max(sum_nums1, sum_nums2) % MOD

c. Increment both i and j. Now, i = 4, j = 3.

We add nums1[i] to f (since j has reached the end of nums2) and increment i. Now, f = 19 + 10 = 29.

Now nums1[i] is 10 and nums2[j] is 9. Since 9 < 10, we add nums2[j] to g and increment j. Now, g = 19 + 9 = 28.

Both pointers have reached the end of their arrays. We take the maximum of f and g, which in this case is f = 29.

So, the maximum score that can be achieved is 29, and we will return this value modulo 10^9 + 7 to get the final answer which is

Solution Implementation

len_nums1, len_nums2 = len(nums1), len(nums2) # Lengths of the input arrays

elif index_nums2 == len_nums2: # nums2 is exhausted, continue with nums1

elif nums1[index_nums1] < nums2[index_nums2]: # Current element of nums1 is smaller</pre>

Update both sums to the maximum of the two sums plus the current element

sum_nums1 = sum_nums2 = max(sum_nums1, sum_nums2) + nums1[index_nums1]

index_nums1 = index_nums2 = 0 # Index pointers for nums1 and nums2

sum_nums1 = sum_nums2 = 0 # Running sums for nums1 and nums2

Move past this common element in both arrays

b. We add the max of f and g to the common value, which gives f = g = 8 + 11 = 19.

class Solution: def maxSum(self, nums1: List[int], nums2: List[int]) -> int:

Process both arrays until we reach the end of one of them while index_nums1 < len_nums1 or index_nums2 < len_nums2:</pre> if index_nums1 == len_nums1: # nums1 is exhausted, continue with nums2 sum_nums2 += nums2[index_nums2] index_nums2 += 1

```
elif nums1[index_nums1] > nums2[index_nums2]: # Current element of nums2 is smaller
else: # Elements in both arrays are equal
```

Python

from typing import List

Initialize variables

```
Java
class Solution {
    public int maxSum(int[] nums1, int[] nums2) {
        final int MODULO = (int) 1e9 + 7; // Define the modulo value as a constant
        int lengthNums1 = nums1.length; // Length of the first array
        int lengthNums2 = nums2.length; // Length of the second array
        int indexNums1 = 0; // Current index in nums1
        int indexNums2 = 0; // Current index in nums2
        long sumNums1 = 0; // Running sum of segments from nums1
        long sumNums2 = 0; // Running sum of segments from nums2
       // While there are elements left to consider in either array
       while (indexNums1 < lengthNums1 || indexNums2 < lengthNums2) {</pre>
            if (indexNums1 == lengthNums1) {
                // If nums1 is exhausted, add remaining nums2 elements to sumNums2
                sumNums2 += nums2[indexNums2++];
            } else if (indexNums2 == lengthNums2) {
                // If nums2 is exhausted, add remaining nums1 elements to sumNums1
                sumNums1 += nums1[indexNums1++];
            } else if (nums1[indexNums1] < nums2[indexNums2]) {</pre>
                // If current element in nums1 is smaller, add it to sumNums1
                sumNums1 += nums1[indexNums1++];
            } else if (nums1[indexNums1] > nums2[indexNums2]) {
                // If current element in nums2 is smaller, add it to sumNums2
                sumNums2 += nums2[indexNums2++];
            } else {
                // If elements are the same, add the max of the two running sums to both sums
                // and move forward in both arrays
                sumNums1 = sumNums2 = Math.max(sumNums1, sumNums2) + nums1[indexNums1];
                indexNums1++;
                indexNums2++;
       // Calculate max of both sums and apply modulo
        int result = (int) (Math.max(sumNums1, sumNums2) % MODULO);
        return result; // Return the result
```

C++

public:

#include <vector>

class Solution {

#include <algorithm>

```
// When both elements are equal, move to the next elements and add the maximum
                // of sum1 and sum2 to both sums.
                sum1 = sum2 = std::max(sum1, sum2) + nums1[index1];
                index1++;
                index2++;
       // Calculate the maximum sum encountered, modulo the defined number.
        return std::max(sum1, sum2) % MODULO;
};
TypeScript
function maxSum(nums1: number[], nums2: number[]): number {
    // Define the modulo value as per the problem's constraint to avoid overflow.
    const mod = 1e9 + 7;
    // Store the lengths of the input arrays.
    const nums1Length = nums1.length;
    const nums2Length = nums2.length;
    // Initialize two variables to keep track of the maximum sum path for each of the arrays.
    let nums1MaxSum = 0;
    let nums2MaxSum = 0;
    // Initialize two pointers for traversing through the arrays.
    let nums1Index = 0:
    let nums2Index = 0;
    // Iterate over the arrays until the end of both is reached.
    while (nums1Index < nums1Length || nums2Index < nums2Length) {</pre>
        if (nums1Index === nums1Length) {
            // If nums1 is exhausted, continue adding the elements from nums2 to nums2's max sum.
           nums2MaxSum += nums2[nums2Index++];
        } else if (nums2Index === nums2Length) {
            // If nums2 is exhausted, continue adding the elements from nums1 to nums1's max sum.
           nums1MaxSum += nums1[nums1Index++];
        } else if (nums1[nums1Index] < nums2[nums2Index]) {</pre>
           // If the current element in nums1 is less than in nums2,
           // add it to the max sum path of nums1.
           nums1MaxSum += nums1[nums1Index++];
        } else if (nums1[nums1Index] > nums2[nums2Index]) {
           // If the current element in nums2 is less than in nums1,
           // add it to the max sum path of nums2.
            nums2MaxSum += nums2[nums2Index++];
        } else {
           // If the elements are equal, add the maximum of the two paths plus the current element,
```

```
while index_nums1 < len_nums1 or index_nums2 < len_nums2:</pre>
    if index_nums1 == len_nums1: # nums1 is exhausted, continue with nums2
        sum_nums2 += nums2[index_nums2]
        index_nums2 += 1
    elif index_nums2 == len_nums2: # nums2 is exhausted, continue with nums1
```

from typing import List

class Solution:

```
sum_nums1 += nums1[index_nums1]
               index_nums1 += 1
           elif nums1[index_nums1] < nums2[index_nums2]: # Current element of nums1 is smaller</pre>
               sum_nums1 += nums1[index_nums1]
               index_nums1 += 1
           elif nums1[index_nums1] > nums2[index_nums2]: # Current element of nums2 is smaller
               sum_nums2 += nums2[index_nums2]
               index_nums2 += 1
           else: # Elements in both arrays are equal
               # Update both sums to the maximum of the two sums plus the current element
               sum_nums1 = sum_nums2 = max(sum_nums1, sum_nums2) + nums1[index_nums1]
               # Move past this common element in both arrays
               index_nums1 += 1
               index_nums2 += 1
       # Return the maximum of the two sums modulo MOD
       return max(sum_nums1, sum_nums2) % MOD
Time and Space Complexity
  The given code aims to find the maximum sum of two non-decreasing arrays where the sum includes each number exactly once,
  and where an element present in both arrays can only be counted in one of the arrays at any intersection point.
```

The time complexity of the code is 0(m + n), where m is the length of nums1 and n is the length of nums2. This is because the

space complexity.

Time Complexity

algorithm uses two pointers i and j to traverse both arrays simultaneously. In the worst case, each pointer goes through the entirety of its corresponding array, resulting in a complete traversal of both arrays. Since the traversal involves constant-time checks and updates, no element is visited more than once, and there are no nested loops, the time complexity is linear with respect to the total number of elements in both arrays. **Space Complexity** The space complexity of the code is 0(1). Aside from the input arrays nums1 and nums2, we only use a constant amount of extra

space for the variables i, j, f, g, and mod. No additional space is allocated that is dependent on the input size, hence the constant