1144. Decrease Elements To Make Array Zigzag

and nums[i] > nums[i + 1], like A[0] < A[1] > A[2] < A[3] > A[4] < ...

Medium <u>Greedy</u> <u>Array</u>

Problem Description

In this problem, we're given an array of integers called nums. Our task is to make the array into a zigzag array with the least number of moves. A zigzag array is defined in one of the following two ways: **Even-index Zigzag:** For every even index i, the element at i is greater than its adjacent elements (nums[i] > nums[i - 1]

- and nums[i] > nums[i + 1]), like A[0] > A[1] < A[2] > A[3] < A[4] > Odd-index Zigzag: For every odd index i, the element at i is greater than its adjacent elements (nums[i] > nums[i - 1]
- A move consists of choosing any element of the array and decreasing its value by 1. The goal is to find out the minimum number of such moves required to achieve a zigzag pattern.

Intuition

To solve this problem, we can approach it by considering the two possible patterns of zigzag separately (even-index zigzag and

We can iterate through every even index i and check if nums[i] is already greater than its adjacent elements. If not, we

Similarly, we iterate through every odd index i doing the same check and calculation for the moves required to transform

odd-index zigzag), then choose the one which requires the fewest moves.

calculate the difference by how much nums [i] needs to be decreased to satisfy the zigzag condition. We add this difference to our total moves for even-index zigzag pattern.

- array nums into an odd-index zigzag pattern. An important point to note is that when checking for the zigzag condition, we only need to ensure that each chosen element (nums[i] for even or odd i) is greater than its adjacent elements if they exist. This implies that for the first and last elements, we
- only consider one adjacent element. After calculating moves for both patterns, we return the minimum of the two totals as the answer. This way, we ensure that we find the zigzag array that requires the least amount of modification (moves).

Solution Approach The solution approach for converting an array nums into a zigzag array is as follows:

We utilize a list ans with two elements initialized to 0. These two elements represent the minimum number of moves required for turning nums into an even-index zigzag array (first element of ans) and an odd-index zigzag array (second element of

We iterate through the array twice, first starting at index 0 for even-index zigzag array, and then starting at index 1 for odd-

For each iteration (for even and odd patterns), we loop through the appropriate indices (i for even and i+1 for odd) with

index zigzag array. This aligns with the two possible zigzag patterns.

ans), respectively.

1], which is nums[j] - nums[j + 1] + 1.

one side is considered if j is at the boundary of the array.

- step 2, to only look at the elements that should be larger than their neighbors. At each selected index j, we calculate the amount to decrease the current element nums[j] to make it lower than both of its neighbors if they exist. This is done with two if conditions:
- If j is not the first element (j > 0), we calculate the required decrease to ensure nums[j] is greater than nums[j 1], which is nums[j] - nums[j - 1] + 1. If j is not the last element (j < n - 1), we calculate the required decrease to ensure nums[j] is greater than nums[j +
- We accumulate the total decreases in ans[i] for each iteration pattern (0 for even-indexed and 1 for odd-indexed zigzag). After the loops, ans contains two numbers: the total moves to make the nums array a zigzag array by promoting even-

indexed elements and the total moves to make it a zigzag array by promoting odd-indexed elements.

zigzag array, regardless of whether we are following the even-indexed or odd-indexed pattern.

element comparison to meet the zigzag pattern requirements with a minimal number of moves.

The amount d is the maximum of the decreases required for both sides (left and right) of the current element nums [j]. Only

The final result is the minimum of the values in lans, which represents the minimum number of moves to convert nums into a

- This approach efficiently computes the desired outcome using array manipulation and condition checking without the need for complex data structures or advanced algorithms. The simplicity of this solution lies in the clever use of iteration, conditionals, and
- Let's consider a small example array nums:

We want to convert this array into either an even-index zigzag or an odd-index zigzag array in the least number of moves possible, following the solution approach described earlier.

• At index 0, nums [0] is 4. Since there is no element at -1, we only need to compare it to nums [1]. No moves are required here because 4 > 3.

• At index 4, nums [4] is 2. Since there is no element at index 5, we only compare with nums [3]. We need to decrease nums [3] (6) to at least 1

Starting with index 0 (even), the selected indices will be 0, 2, 4, hence we check and modify only these indices.

• At index 2, nums [2] is 7. It is already greater than both its neighbors nums [1] and nums [3]. No moves required.

for it to be smaller than nums[4]. Therefore, 6 - 2 + 1 is 5 moves required.

Now, starting with index 1 (odd), the selected indices to check will be 1, 3.

number of moves required to make the given nums a zigzag array is 0.

Initialize a list to store the number of moves for two scenarios:

Loop over the two scenarios: starting with the first and second elements

If not the first element, compare with the left neighbor

difference = max(difference, nums[j] - nums[j + 1] + 1)

// Function to calculate the minimum moves to make the array 'nums' a zigzag sequence.

// ans[0] is for the scenario where the first element is less than the second.

// ans[1] is for the scenario where the first element is greater than the second.

// Outer loop for the two possible zigzag patterns (starting with lower or higher).

// Inner loop to iterate over the array with step of 2 to maintain zigzag.

decrement = std::max(decrement, nums[i] - nums[i - 1] + 1);

decrement = std::max(decrement, nums[i] - nums[i + 1] + 1);

// Check if there is a previous element, and if so, make 'nums[i]' less than 'nums[i - 1]'.

// Check if there is a next element, and if so, make 'nums[i]' less than 'nums[i + 1]'.

// Add the required decrement to maintain the zigzag pattern for this position to the answer.

Return the minimum number of moves required between the two scenarios

1. Making the first element higher than its neighbors (odd indexed after the first)

Now, check the elements that need to be adjusted according to the scenario

Initialize the variable to record the number of moves to make the

Adiust the number of moves to ensure nums[i] is less than nums[j + 1]

Add the calculated moves for this element to the total for this scenario

2. Making the second element higher than its neighbors (even indexed after the first)

Total moves required for the even-index zigzag is 0 + 0 + 5 = 5.

• At index 1, nums[1] is 3. We need to reduce it to 3 to less than nums[0] (4). Hence no moves required as 4 > 3.

needed.

Transforming into Odd-Index Zigzag

Solution Implementation

moves_required = [0, 0]

num_elements = len(nums)

for index in range(2):

Get the length of the input list

difference = 0

return min(moves_required)

Java

C++

public:

#include <vector>

class Solution {

#include <algorithm>

int movesToMakeZigzag(std::vector<int>& nums) {

std::vector<int> ans(2, 0);

// Size of the input array.

int decrement = 0;

 $if (i > 0) {$

if (i < n - 1) {

return std::min(ans[0], ans[1]);

function movesToMakeZigzag(nums: number[]): number {

ans[pattern] += decrement;

// Return the minimum of the two answers.

int n = nums.size();

// Initializing answer array for the two scenarios.

for (int pattern = 0; pattern < 2; ++pattern) -</pre>

for (int i = pattern; i < n; i += 2) {</pre>

// Increment the decrement count appropriately.

// Increment the decrement count appropriately.

// Function to calculate the minimum number of moves required to form a zigzag sequence

difference = max(difference, nums[j] - nums[j - 1] + 1)

difference = max(difference, nums[j] - nums[j + 1] + 1)

Adiust the number of moves to ensure nums[i] is less than nums[i + 1]

Add the calculated moves for this element to the total for this scenario

If not the last element, compare with the right neighbor

Return the minimum number of moves required between the two scenarios

// Initialize an array to store the minimum moves for both patterns

Example Walkthrough

nums = [4, 3, 7, 6, 2]

Transforming into Even-Index Zigzag

Total moves required for the odd-index zigzag is 0 + 0 = 0.

Conclusion

Comparing both patterns, odd-index zigzag requires 0 moves and even-index zigzag requires 5 moves. Hence, the minimum

Using the solution approach, we can thus determine that the original array is already an odd-index zigzag array and no moves are

• At index 3, nums[3] is 6. It should be greater than nums[2] (7) but it's not. So, we need to decrease nums[2] to 5 (nums[3] - nums[2] + 1),

which is 6 - 7 + 1 = 0 moves. Additionally, nums [3] should also be greater than nums [4] (2), but it already is, so no further moves required.

Python class Solution:

if i > 0: # Calculate the required moves to ensure nums[i] is less than nums[j - 1] difference = max(difference, nums[j] - nums[j - 1] + 1) # If not the last element, compare with the right neighbor

moves_required[index] += difference

if j < num elements - 1:

for i in range(index, num elements, 2):

current element lower than its neighbors

def movesToMakeZigzag(self, nums: List[int]) -> int:

```
class Solution {
   public int movesToMakeZigzag(int[] nums) {
       int[] movesRequired = new int[2]; // This array will hold the moves required for the two cases.
       int n = nums.length; // The length of the input array.
       // Loop through the two different cases:
       // Case 0: Making the even—indexed elements lower ('Zig')
       // Case 1: Making the odd-indexed elements lower ('Zag')
       for (int caseIndex = 0; caseIndex < 2; ++caseIndex) {</pre>
           // Inner loop to go through elements based on the current case:
           // Either start from index 0 for even ('Zig'), or index 1 for odd ('Zag')
           for (int currentIndex = caseIndex; currentIndex < n; currentIndex += 2) {</pre>
               int decrement = 0; // This will store the required decrement for the current element.
               // If not the first element, check with the previous element
               if (currentIndex > 0) {
                    decrement = Math.max(decrement, nums[currentIndex] - nums[currentIndex - 1] + 1);
               // If not the last element, check with the next element
               if (currentIndex < n - 1) {</pre>
                    decrement = Math.max(decrement, nums[currentIndex] - nums[currentIndex + 1] + 1);
               // Increment the moves count for the current case by the decrement calculated
               movesRequired[caseIndex] += decrement;
       // Return the minimum of the two cases as the result
       return Math.min(movesRequired[0], movesRequired[1]);
```

const minMoves: number[] = Array(2).fill(0); // Get the length of the nums array const length = nums.length; // Loop for odd and even indices separately

};

TypeScript

```
for (let pattern = 0; pattern < 2; ++pattern) {</pre>
       // Start from 0 or 1 index based on the pattern and increment by 2
        for (let index = pattern; index < length; index += 2) {</pre>
            let difference = 0; // Difference needed to make the current element smaller
            // If there is a previous element and it is not less than the current one
            if (index > 0) {
                difference = Math.max(difference, nums[index] - nums[index - 1] + 1);
            // If there is a next element and it is not less than the current one
            if (index < length - 1) {</pre>
                difference = Math.max(difference, nums[index] - nums[index + 1] + 1);
           // Add the required difference to the moves count for the current pattern
            minMoves[pattern] += difference;
   // Return the minimum of the two patterns' moves counts
   return Math.min(...minMoves);
class Solution:
   def movesToMakeZigzag(self, nums: List[int]) -> int:
       # Initialize a list to store the number of moves for two scenarios:
       # 1. Making the first element higher than its neighbors (odd indexed after the first)
       # 2. Making the second element higher than its neighbors (even indexed after the first)
       moves required = [0, 0]
       # Get the length of the input list
       num_elements = len(nums)
       # Loop over the two scenarios: starting with the first and second elements
        for index in range(2):
           # Now. check the elements that need to be adjusted according to the scenario
            for j in range(index, num elements, 2):
                # Initialize the variable to record the number of moves to make the
                # current element lower than its neighbors
                difference = 0
               # If not the first element, compare with the left neighbor
               if i > 0:
                   # Calculate the required moves to ensure nums[i] is less than nums[j - 1]
```

Time and Space Complexity

return min(moves_required)

if j < num elements - 1:</pre>

moves_required[index] += difference

The time complexity of the provided code is O(n), where n is the length of the input list nums. This is because the code contains two nested loops. The outer loop runs twice (once for each of the two possible zigzag configurations), and the inner loop iterates

over every other element of the list. Since at most, each element is considered once per outer loop iteration, the total number of operations is proportional to the size of the input list, resulting in a linear time complexity. **Space Complexity**

Time Complexity

The space complexity is 0(1), meaning it requires constant additional space regardless of the input size. The array ans stores only two elements corresponding to the two possible zigzag configurations, making its size fixed. The variables d and n also do not depend on the input size, so the memory usage does not grow with the size of nums.