1846. Maximum Element After Decreasing and Rearranging

Problem Description

Medium

The problem presents an array of positive integers named arr. The objective is to apply certain operations to modify the array so that it meets the following criteria:

Greedy Array

- ranges from 1 to arr. length 1), we have |arr[i] arr[i 1]| <= 1, with |x| representing the absolute value of x. We are allowed to perform two types of operations as many times as needed:

2. The absolute difference between any two consecutive elements in the array should be less than or equal to 1. That is, for each i (where i

 Decrease the value of any element to a lower positive integer. Rearrange the elements in any order.

The goal is to return the maximum possible value of an element in arr after performing these operations to satisfy the given

Sorting

conditions.

1. The first element of the array should be 1.

Intuition The intuition behind the solution is based on the understanding that in order to minimize the absolute differences between

a sequence where each element is either the same or 1 greater than its predecessor.

Here is the approach we can follow to arrive at a solution: Sort the array in non-decreasing order. This is because rearranging the elements can only help if they are sorted, which ensures that no two adjacent elements have an absolute difference greater than allowed after the operations.

adjacent elements, it makes sense to sort the array in non-decreasing order. Once sorted, the elements of the array should form

Loop through the array starting from the second element. For each i from 1 to arr.length - 1, we need to ensure the absolute difference condition |arr[i] - arr[i - 1]| <= 1. The best way to achieve this without violating the array's non-decreasing

order and satisfies the adjacent element condition.

Set the first element of arr to 1, since that's a mandatory requirement.

After completing this process, the last element in the array would hold the maximum value possible while satisfying the required conditions. The code provided gives us a straightforward implementation of this approach.

order is to check if arr[i] is more than arr[i - 1] + 1 and if so, decrease it to arr[i - 1] + 1. This keeps the sequence in

implementation works in detail:

The provided solution utilizes a simple sorting-based approach with a single pass modification after sorting. Here's how the

Setting First Element: Per the problem requirements, the first element must be set to 1, so without considering its previous

already set to 1. For each element arr[i], we want to ensure that it is not more than 1 greater than the previous element

Returning Result: After the loop concludes, the array now satisfies all the given conditions. Since the array elements have

been sorted and then properly adjusted, the maximum value that adheres to the constraints is located at the end of the array.

Sorting: The first step is to sort the array arr in non-decreasing order. This is done with arr.sort(). Sorting is critical because it places elements in a sequence that makes it easier to minimize absolute differences between consecutive

Hence, max(arr) returns the maximum value from arr.

expensive operation in the provided solution.

retrieving the maximum possible element value.

between consecutive elements is at most 1.

Each element is at most 1 greater than the previous one.

Let's walk through a small example to illustrate the solution approach:

value of 4 is too high, so we decrease it: arr = [1, 2, 7].

def maximumElementAfterDecrementingAndRearranging(self, arr: List[int]) -> int:

First, sort the array to make it easier to apply the decrementing rule.

The first element must be set to 1 according to the problem constraints.

Calculate the maximum decrement for the current element so that

Return the last element in the modified array, which is the maximum.

// Step 2: The first element should be 1 as per problem's constraint

// Step 3: Iteratively check each element starting from the second one

// to ensure it's not more than 1 greater than its predecessor

int decrement = Math.max(0, arr[i] - arr[i - 1] - 1);

// Update the maximumElement if current is greater

maximumElement = Math.max(maximumElement, arr[i]);

int maximumElementAfterDecrementingAndRearranging(vector<int>& arr) {

// Calculate the decrement value needed while ensuring it's not negative.

const decrementValue = Math.max(0, arr[i] - arr[i - 1] - 1);

// Return the maximum element found after performing the operations.

def maximumElementAfterDecrementingAndRearranging(self, arr: List[int]) -> int:

First, sort the array to make it easier to apply the decrementing rule.

The first element must be set to 1 according to the problem constraints.

Calculate the maximum decrement for the current element so that

Apply the calculated decrement to ensure each element is at most

Iterate over the sorted array starting from the second element.

it is not greater than the preceding element plus one.

decrement_value = max(0, arr[i] - arr[i - 1] - 1)

1 greater than the previous element.

// Decrement the current element by the calculated value.

maxElement = Math.max(maxElement, arr[i]);

arr[i] -= decrementValue;

return maxElement;

from typing import List

arr.sort()

arr[0] = 1

class Solution:

// We subtract 1 to allow only increments by 1 from the previous number or remaining unchanged.

// Update 'maxElement' to be the higher value between the current 'maxElement' and the new value of arr[i].

// Calculate the amount by which we can decrement the current element

// Step 4: Return the highest value found which satisfies the given conditions

// Ensure that we do not decrement the value into negatives by using Math.max with 0

// Initialize the answer with the first element's value

// Step 1: Sort the input array to check the difference between consecutive elements

return arr[-1] # Use -1 to access the last element for clarity.

Iterate over the sorted array starting from the second element.

it is not greater than the preceding element plus one.

 $decrement_value = max(0, arr[i] - arr[i - 1] - 1)$

is too high, so we decrease it: arr = [1, 2, 3].

complexity as it avoids recursive calls or additional passes through the array.

value, we directly assign arr[0] = 1. **Loop Through Elements:** We then iterate over the array, starting from the second element (index 1), since the first element is

•

elements.

arr[i - 1].

Data Structures and Patterns:

Algorithm:

Solution Approach

- This is done by calculating the difference d between arr[i] and arr[i 1] + 1 and then checking if this is greater than 0. If it is, it means that arr[i] is too large and needs to be reduced. The element arr[i] is then decreased by d, ensuring that the current element arr[i] is either equal to or one more than arr[i - 1].
- Array/Sorting: The solution uses the given array arr as the primary data structure. No additional complex data structures are needed. The sorting algorithm used by Python's sort() function fundamentally determines the overall efficiency of this approach.

Iterative Loop: An iterative loop is utilized to adjust the values of arr after the initial sort. This reduces the algorithm's

modification of arr. This is efficient both in terms of space and time. **Computational Complexity:**

Time Complexity: 0(n log n), where n is the number of elements in arr. Sorting the array is the most computationally

Space Complexity: 0(1), as we are operating in-place and not using extra space that is dependent on the input size.

This implementation yields the desired result by sorting the array, adjusting its values while maintaining sorted order, and finally,

In-Place Modifications: All operations are performed in place, which means no additional arrays are created during the

Example Walkthrough

Given the input array arr = [4, 7, 2], we want to modify it following our operation rules so that the first element is 1 and the

absolute difference between consecutive elements is at most 1. We will demonstrate each step of the algorithm with this array.

Setting First Element: Next, we must set the first element to 1 to satisfy the problem's conditions. Set the first element: arr =

For the second element (arr[1]), the previous element is 1, so the second element should not be greater than 2. The

For the third element (arr[2]), the previous element is 2, so the third element should not be greater than 3. The value of 7

[1, 4, 7]. Loop Through Elements: Now, we start at the second element and iterate through the array to ensure the absolute difference

Sorting: First, we sort the array in non-decreasing order. Sort the array: arr = [2, 4, 7].

Returning Result: After modifying the array, the maximum value while satisfying the conditions is the last element. Return the max value: max(arr) = 3.

The first element is 1.

from typing import List

arr.sort()

arr[0] = 1

Arrays.sort(arr);

int maximumElement = 1;

for (int i = 1; i < arr.length; ++i) {</pre>

// Decrement the current element

arr[i] -= decrement;

return maximumElement;

arr[0] = 1;

for i in range(1, len(arr)):

class Solution:

that:

Python

Java

the criteria, and no elements are more than 1 apart consecutively. Solution Implementation

Finally, we returned 3 as the maximum possible value after performing the allowed operations. The sorted array arr now satisfies

By following the steps above, we have successfully modified the original array [4, 7, 2] to [1, 2, 3] meeting the conditions

- # Apply the calculated decrement to ensure each element is at most # 1 greater than the previous element. arr[i] -= decrement_value
- class Solution { public int maximumElementAfterDecrementingAndRearranging(int[] arr) {

C++ class Solution {

public:

```
// Step 1: Sort the array in non-decreasing order
        sort(arr.begin(), arr.end());
       // Step 2: The first element should be set to 1 according to the problem statement
        arr[0] = 1;
        // Initialize the answer with the first element's value
        int maximumElement = 1;
        // Step 3: Iterate over the sorted array starting from the second element
        for (int i = 1; i < arr.size(); ++i) {</pre>
            // Determine the maximum decrement needed to maintain non-decreasing order
            // It should not be negative, hence max with 0
            int decrement = max(0, arr[i] - arr[i - 1] - 1);
            // Decrease the current element to make the sequence non-decreasing if needed
            arr[i] -= decrement;
            // Update the maximum element encountered so far
            maximumElement = max(maximumElement, arr[i]);
        // Return the largest possible element after performing the operations
        return maximumElement;
};
TypeScript
function maximumElementAfterDecrementingAndRearranging(arr: number[]): number {
   // Sort the input array in non-decreasing order.
    arr.sort((a, b) => a - b);
    // The first element should always be set to 1.
    arr[0] = 1;
    // Initialize 'maxElement' to keep track of the maximum element after operations.
    let maxElement = 1;
    // Iterate over the array starting from the second element.
    for (let i = 1; i < arr.length; ++i) {</pre>
```

Return the last element in the modified array, which is the maximum. return arr[-1] # Use -1 to access the last element for clarity. Time and Space Complexity

Time Complexity

for i in range(1, len(arr)):

arr[i] -= decrement_value

sorting algorithm derived from merge sort and insertion sort. The worst-case time complexity of the sort operation is 0(n log n), where n is the number of elements in the array.

The time complexity of the given code is determined by the main operations it performs:

- Iterating over the Sorted Array: After sorting, the function iterates through the sorted array once to adjust the values. This is a linear operation with a time complexity of O(n). Since sorting the array dominates the overall time complexity, the final time complexity of the entire function is 0(n log n).
- **Space Complexity** The space complexity of the code is analyzed based on the extra space used in addition to the input array:

Sorting the Array: The in-place sorting does not require additional space, so it uses 0(1) additional space.

Sorting the Array: The sort operation on the array is the most time-consuming part. Python uses TimSort, which is a hybrid

Iteration: No additional data structures are used during iteration, so it only requires constant space.

From the above analysis, the space complexity of the function is 0(1), which means it only requires a constant amount of extra space.