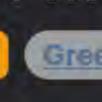
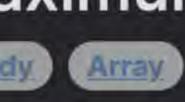
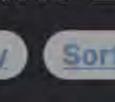
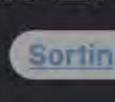
1846. Maximum Element After Decreasing and Rearranging

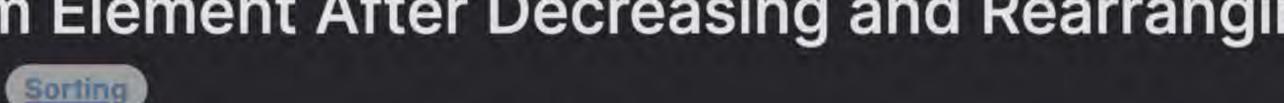












Leetcode Link

The problem presents an array of positive integers named arr. The objective is to apply certain operations to modify the array so

Problem Description

that it meets the following criteria: The first element of the array should be 1.

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

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

Decrease the value of any element to a lower positive integer.

conditions.

Intuition

that no two adjacent elements have an absolute difference greater than allowed after the operations.

elements, it makes sense to sort the array in non-decreasing order. Once sorted, the elements of the array should form 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: 1. Sort the array in non-decreasing order. This is because rearranging the elements can only help if they are sorted, which ensures

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

satisfies the adjacent element condition.

- 3. 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 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 order and
- 4. After completing this process, the last element in the array would hold the maximum value possible while satisfying the required

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

- The code provided gives us a straightforward implementation of this approach. Solution Approach
- The provided solution utilizes a simple sorting-based approach with a single pass modification after sorting. Here's how the implementation works in detail:

1. 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

Algorithm:

places elements in a sequence that makes it easier to minimize absolute differences between consecutive elements. 2. Setting First Element: Per the problem requirements, the first element must be set to 1, so without considering its previous

value, we directly assign arr[0] = 1.

max(arr) returns the maximum value from arr.

conditions.

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 arr[i -1].

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

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

3. Loop Through Elements: We then iterate over the array, starting from the second element (index 1), since the first element is

current element arr[i] is either equal to or one more than arr[i - 1]. 4. Returning Result: After the loop concludes, the array now satisfies all the given conditions. Since the array elements have been

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

Data Structures and Patterns: 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

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

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

modification of arr. This is efficient both in terms of space and time.

Computational Complexity:

Example Walkthrough

retrieving the maximum possible element value.

between consecutive elements is at most 1.

- Time Complexity: 0(n log n), where n is the number of elements in arr. Sorting the array is the most computationally expensive operation in the provided solution.
- Let's walk through a small example to illustrate the solution approach:

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

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

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

• 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,

2. Setting First Element: Next, we must set the first element to 1 to satisfy the problem's conditions. Set the first element: arr = [1, 4, 7].

3. Loop Through Elements: Now, we start at the second element and iterate through the array to ensure the absolute difference

For the second element (arr[1]), the previous element is 1, so the second element should not be greater than 2. The value of 4 is too high, so we decrease it: arr = [1, 2, 7].

. The first element is 1.

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

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

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

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.

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

public int maximumElementAfterDecrementingAndRearranging(int[] arr) {

int maximumElementAfterDecrementingAndRearranging(vector<int>& arr) {

// Step 2: The first element should be set to 1 according to the problem statement

// Determine the maximum decrement needed to maintain non-decreasing order

// Decrease the current element to make the sequence non-decreasing if needed

// Step 3: Iterate over the sorted array starting from the second element

// Return the largest possible element after performing the operations

// Step 1: Sort the array in non-decreasing order

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

// It should not be negative, hence max with 0

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

// Update the maximum element encountered so far

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

sort(arr.begin(), arr.end());

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

int maximumElement = 1;

arr[i] -= decrement;

return maximumElement;

Iterate over the sorted array starting from the second element.

it is not greater than the preceding element plus one.

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

max value: max(arr) = 3.By following the steps above, we have successfully modified the original array [4, 7, 2] to [1, 2, 3] meeting the conditions that:

4. Returning Result: After modifying the array, the maximum value while satisfying the conditions is the last element. Return the

Python Solution from typing import List

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

16 # Apply the calculated decrement to ensure each element is at most 17 # 1 greater than the previous element. arr[i] -= decrement_value 20

Java Solution

class Solution {

class Solution:

10

12

13

14

15

18

19

21

22

23

arr.sort()

arr[0] = 1

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

```
// Step 1: Sort the input array to check the difference between consecutive elements
           Arrays.sort(arr);
           // Step 2: The first element should be 1 as per problem's constraint
           arr[0] = 1;
10
           // Initialize the answer with the first element's value
12
           int maximumElement = 1;
13
           // Step 3: Iteratively check each element starting from the second one
14
15
           for (int i = 1; i < arr.length; ++i) {</pre>
16
               // Calculate the amount by which we can decrement the current element
               // to ensure it's not more than 1 greater than its predecessor
               // Ensure that we do not decrement the value into negatives by using Math.max with 0
19
               int decrement = Math.max(0, arr[i] - arr[i - 1] - 1);
20
21
22
               // Decrement the current element
               arr[i] -= decrement;
24
25
               // Update the maximumElement if current is greater
26
               maximumElement = Math.max(maximumElement, arr[i]);
27
28
29
           // Step 4: Return the highest value found which satisfies the given conditions
           return maximumElement;
30
31
32 }
33
```

29 }; 30

C++ Solution

1 class Solution {

arr[0] = 1;

2 public:

14

15

16

20

21

23

24

25

26

27

28

```
Typescript Solution
   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;
10
       // Iterate over the array starting from the second element.
       for (let i = 1; i < arr.length; ++i) {</pre>
           // Calculate the decrement value needed while ensuring it's not negative.
13
           // We subtract 1 to allow only increments by 1 from the previous number or remaining unchanged.
14
           const decrementValue = Math.max(0, arr[i] - arr[i - 1] - 1);
15
16
           // Decrement the current element by the calculated value.
           arr[i] -= decrementValue;
19
           // Update 'maxElement' to be the higher value between the current 'maxElement' and the new value of arr[i].
20
           maxElement = Math.max(maxElement, arr[i]);
21
22
24
       // Return the maximum element found after performing the operations.
       return maxElement;
25
26 }
27
```

Time Complexity

Time and Space Complexity

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. 2. Iterating over the Sorted Array: After sorting, the function iterates through the sorted array once to adjust the values. This is a

The space complexity of the code is analyzed based on the extra space used in addition to the input array:

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

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 $0(n \log n)$.

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

Space Complexity

- 1. Sorting the Array: The in-place sorting does not require additional space, so it uses 0(1) additional space. 2. 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.