1299. Replace Elements with Greatest Element on Right Side



Leetcode Link

Problem Description

In the given problem, we have an array called arr. The task is to modify each element in this array, so that for each position in the array, you replace its value with the greatest value of the elements positioned to its right in the array. For the last element, since there are no elements to its right, you replace it with -1. Finally, the modified array is returned.

Intuition

The intuitive approach to solving this problem is to start from the rightmost end of the array and move leftward. This way, at any given position i, we will have already processed the elements to its right and thus would know the greatest element among them.

- 1. We initialize a variable m to -1, which will keep track of the maximum value we've encountered as we iterate the array from the right to left.
- 2. We start iterating from the end of the array.
- 3. For the current element at index i, we store its value temporarily in a variable t since it will be overwritten and we need to compare it against m afterward.
- 4. The current element at index i is then replaced with the maximum value m found so far.
- 5. We update m to be the maximum of the old maximum m and the value t we saved before overwriting the element at i.
- 6. We repeat these steps until all elements have been processed and return the modified array.

By the end of this process, each element is replaced by the maximum value to its right, and the last element has been correctly replaced with -1.

Solution Approach

The solution uses a simple iterative approach without any complex data structures or additional space requirement (besides the input array). Here's a step-by-step explanation of the implementation details:

- 1. Initialize a variable m as -1. This variable serves as a placeholder for the greatest element found so far to the right of the current index during the backward traversal of the array.
- 2. Start a loop to traverse the array backwards. The loop starts from the last index, which is len(arr) 1, moving towards the first index 0. We go backwards so that the rightmost elements get processed first.
- 3. In each iteration, capture the current element before replacing it. The variable t = arr[i] is used to store the current value of the array at index i.
- 4. In the next step, we replace the current element at index i with the greatest element on its right m.
- 5. The last action in the iteration updates m to the maximum value between the previous maximum m and the value we just replaced (t). We use the built-in max function for this purpose: m = max(m, t).
- 6. Continue this process until the loop finishes (all elements are checked).
- 7. Once the loop finishes, the modified array gets returned directly.

This is a linear time complexity O(n) solution, where n is the number of elements in the array. No additional space is used, making the space complexity 0(1).

right in a single pass, making it a highly efficient in-place algorithm.

It's important to note that the implementation effectively overwrites each element in the array with the greatest value found to its

Let's consider an example array arr = [4, 3, 2, 1] to illustrate the solution approach described above.

Example Walkthrough

2. Start traversing the array backwards:

efficient solution with a time complexity of O(n) and a space complexity of O(1).

// Iterate through the array from the end to the beginning

// Iterate through the array from the back to the front

int currentValue = arr[i]; // Store the current value temporarily

for (int i = arr.size() - 1; i >= 0; --i) {

for (int i = arr.length - 1; i >= 0; --i) {

Reverse iterate through the array

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

1. Initialize m as -1. This will hold the maximum value found to the right of the current element.

```
\circ For i = 3 (last index), arr[i] is 1. We store this in t, and then we set arr[i] to -1, since it is the last element.
     • Update m to max(m, t), which is max(-1, 1), so m becomes 1.
3. Move to i = 2:
    arr[i] is 2. Store this in t, then set arr[i] to m, which is 1.

    Update m to max(m, t), which is max(1, 2), so m becomes 2.

4. Move to i = 1:
    arr[i] is 3. Store this in t, then set arr[i] to m, which is 2.

    Update m to max(m, t), which is max(2, 3), so m becomes 3.

5. Finally, move to i = 0:
    arr[i] is 4. Store this in t, then set arr[i] to m, which is 3.

    Update m to max(m, t), which is max(3, 4), but since this is the last iteration, the value of m is no longer needed.

6. The loop has finished, and the modified array is now arr = [3, 2, 1, -1].
```

Python Solution

The algorithm has successfully replaced each element with the maximum value among the elements positioned to its right, and the

last element has been replaced with -1. The whole process required only a single pass through the input array, making it a very

def replaceElements(self, arr: List[int]) -> List[int]: # Initialize the maximum value found to the right of the current element $max_to_the_right = -1$

class Solution:

```
# Store the current element before updating
               current_element = arr[i]
10
               # Replace the current element with the maximum value found so far
11
12
               arr[i] = max_to_the_right
13
14
               # Update the maximum value by comparing it with the previously stored current element
               max_to_the_right = max(max_to_the_right, current_element)
17
           # Return the modified array
           return arr
19
   # The List type should be imported from the 'typing' module
   from typing import List
22
Java Solution
   class Solution {
       public int[] replaceElements(int[] arr) {
           // Start from the end of the array
```

int maxSoFar = -1; // Initially set the maximum to -1 since it will be the replacement for the last element

int currentValue = arr[i]; // Store the current value to temporarily hold it

arr[i] = maxSoFar; // Replace the current element with the max of elements to its right maxSoFar = Math.max(maxSoFar, currentValue); // Update the maxSoFar with the maximum between the maxSoFar and currentValue

```
13
           // Return the modified array
14
           return arr;
15
16 }
17
C++ Solution
 1 #include <vector>
2 #include <algorithm>
   class Solution {
   public:
       std::vector<int> replaceElements(std::vector<int>& arr) {
           int maxSoFar = -1; // Initially set the maximum to -1 since it will be the replacement for the last element
```

maxSoFar = std::max(maxSoFar, currentValue); // Update maxSoFar to the maximum of maxSoFar and currentValue

// Return the modified array 16 return arr; 18

10

13

14

15

15

```
19 };
20
Typescript Solution
 1 // Function to replace each element with the greatest element on its right side
   function replaceElements(arr: number[]): number[] {
       // Start from the end of the array
       let maxSoFar: number = −1; // Initially set the max to −1 because it will be the replacement for the last element
       // Iterate through the array from the end to the start
       for (let i = arr.length - 1; i >= 0; --i) {
           // Temporarily hold the current value of the element
           let currentValue: number = arr[i];
           // Replace the current element with the greatest value to its right
10
           arr[i] = maxSoFar;
11
           // Update the maxSoFar to the maximum between the maxSoFar and currentValue
12
           maxSoFar = Math.max(maxSoFar, currentValue);
14
```

arr[i] = maxSoFar; // Replace the current element with the max of elements to its right

// Return the modified array with replaced elements 16 return arr;

Time Complexity The provided function replaceElements consists of a single loop that iterates backwards through the input list arr. On each iteration, the function updates the current element with the maximum value found in the elements to the right of the current element. This loop runs exactly n times where n is the length of the list. Thus, the time complexity of the algorithm is O(n), where n is the number

of elements in arr.

Space Complexity

additional space regardless of the input size.

Time and Space Complexity