665. Non-decreasing Array

Medium Array

Problem Description

This problem requires us to assess whether a given array `nums` containing `n` integers can be converted into a non-decreasing array by altering no more than one element. An array is regarded as non-decreasing if for every index `i`, where `i` ranges from `0` to `n - 2`, the following condition is satisfied: `nums[i] <= nums[i + 1]`. The challenge lies in determining whether we can achieve such a state with a single modification or if the array's current state is already non-decreasing.

Intuition

To solve this problem, we must iterate through the array and identify any pair of consecutive elements where the first is greater than the second (`nums[i] > nums[i + 1]`). This condition indicates that the array is not non-decreasing at that point. When we find such a pair, we have two possible actions to try:

1. Lower the first element (`nums[i]`) to match the second one (`nums[i + 1]`), which can help maintain a non-decreasing order if the rest of the array is non-decreasing.
2. Alternatively, raise the second element (`nums[i + 1]`) to match the first one (`nums[i]`), which again can make the entire array non-decreasing if the rest of it adheres to the rule.

For both scenarios, after making the change, we need to check if the entire array is non-decreasing. If it is, then we can achieve the goal with a single modification. If we make it through the entire array without needing more than one change, the array is either already non-decreasing or can be made so with one modification. The key here is recognizing that if there are two places where `nums[i] > nums[i + 1]`, we can't make the array non-decreasing with just a single change.

Solution Approach

The solution employs a straightforward iterative approach along with a helper function `is_sorted`, which checks if the array passed to it is non-decreasing. This is done using the `pairwise` utility to traverse the array in adjacent pairs and ensuring each element is less than or equal to the next.

Here's a step-by-step breakdown of the algorithm:

1. The function `checkPossibility` begins by iterating over the array `nums` using a loop that runs from the starting index to the second-to-last index of the array.
2. In each iteration, it compares the current element `nums[i]` with the next element `nums[i + 1]`. If it finds these elements are ordered correctly (`nums[i] <= nums[i + 1]`), it continues to the next pair; otherwise, it indicates a potential spot for correction.
3. When a pair is found where `nums[i] > nums[i + 1]`, the solution has two possibilities for correction:
   - **Lower the first element**: It sets `nums[i] = nums[i + 1]` to match the second element and uses the `is_sorted` function to check if this change makes the entire array non-decreasing. If so, it returns `True`.
   - **Raise the second element**: If the previous step does not yield a non-decreasing array, the solution resets `nums[i]` to its original value and sets `nums[i + 1] = nums[i]`. Then it checks with the `is_sorted` function once more. If the array is non-decreasing after this change, it returns `True`.
4. Throughout the iteration, if no change is needed or if the change leads to a non-decreasing array, the function moves to the next pair of elements.
5. In a scenario where no pairs violate the non-decreasing order, the function will complete the loop and return `True`, as no alteration is required or a maximum of one was sufficient.

In this algorithm, the data structure used is the original input array `nums`. The pattern employed here revolves around validating an array's non-decreasing property and the ability to stop as soon as the requirement is violated twice since that indicates more than one change would be necessary. The elegance of this solution lies in its `O(n)` time complexity, as it requires only a single pass through the array to determine its non-decreasing property with at most one modification.

In summary, the provided solution correctly handles both the detection of a non-decreasing sequence violation and the decision-making for adjusting the array, efficiently reaching a verdict with minimal changes and checks.

Example Walkthrough

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

Suppose we have an array `nums = [4, 2, 3]`. According to the problem, we are allowed to make at most one modification to make the array non-decreasing. Let's process this array according to the algorithm described above.

1. We start by checking the array from left to right. We compare the first and second elements: `4` and `2`. Since `4` is greater than `2`, this violates the non-decreasing order.
2. Now, we have two possibilities to correct the array:
   - We can lower the first element `4` to `2`, making the array `[2, 2, 3]`.
   - We can raise the second element `2` to `4`, making the array `[4, 4, 3]`.
3. Let's try the first possibility. We lower `4` to `2`. Now we need to check if after this modification, the array is non-decreasing. By simple inspection, we see `[2, 2, 3]` is indeed a non-decreasing array.
4. Since the array `[2, 2, 3]` is non-decreasing, we return `True`, indicating that the original array `[4, 2, 3]` can be made non-decreasing with a single modification.

The algorithm successfully finds the correct modification on the first try, and further evaluation is not necessary. The solution is efficient and adheres to `O(n)` time complexity, as it processes each element of the array only once.

Solution Implementation

Python

```python
from typing import List


class Solution:
    def check_possibility(self, nums: List[int]) -> bool:
        # Helper function to check if the list is non-decreasing.
        def is_sorted(arr: List[int]) -> bool:
            for i in range(len(arr) - 1):
                if arr[i] > arr[i+1]:
                    return False
            return True

        n = len(nums)
        # Go through each pair in the list.
        for i in range(n - 1):
            # If the current element is greater than the next element, a modification is needed.
            if nums[i] > nums[i + 1]:
                # Temporarily change the current element to the next one and check if sorted.
                temp = nums[i]
                nums[i] = nums[i + 1]
                if is_sorted(nums):
                    return True
                # If not sorted, revert the change and modify the next element instead.
                nums[i] = temp
                nums[i + 1] = temp
                return is_sorted(nums)
        # If no modification needed, then it is already non-decreasing.
        return True


# Example usage:
# sol = Solution()
# print(sol.check_possibility([4,2,3])) # True
# print(sol.check_possibility([4,2,1])) # False
```

Java

```java
class Solution {

    // Main method to check if the array can be made non-decreasing by modifying at most one element
    public boolean checkPossibility(int[] nums) {
        // Iterate through the array elements
        for (int i = 0; i < nums.length - 1; ++i) {
            // Compare current element with the next one
            int current = nums[i];
            int next = nums[i + 1];

            // If the current element is greater than the next, we need to consider modifying one of them
            if (current > next) {
                // Temporarily modify the current element to the next element's value
                nums[i] = next;
                // Check if the array is sorted after this modification
                if (isSorted(nums)) {
                    return true;
                }
                // Revert the change to the current element
                nums[i] = current;
                // Permanently modify the next element to the current element's value
                nums[i + 1] = current;
                // Return whether the array is sorted after this second modification
                return isSorted(nums);
            }
        }

        // If no modifications were needed, the array is already non-decreasing
        return true;
    }

    // Helper method to check if the array is sorted in non-decreasing order
    private boolean isSorted(int[] nums) {
        // Iterate through the array elements
        for (int i = 0; i < nums.length - 1; ++i) {
            // If the current element is greater than the next, the array is not sorted
            if (nums[i] > nums[i + 1]) {
                return false;
            }
        }
        // If no such pair is found, the array is sorted
        return true;
    }
}
```

C++

```cpp
#include <vector>
#include <algorithm>

class Solution {
public:
    // Function to check if it's possible to make the array non-decreasing
    // by modifying at most one element.
    bool checkPossibility(std::vector<int>& nums) {
        int n = nums.size(); // Get the size of the input array

        // Iterate over the array to find a pair where the current element
        // is greater than the next element.
        for (int i = 0; i < n - 1; ++i) {
            int current = nums[i], next = nums[i + 1];

            // If a pair is found where the current element is greater than the next,
            // we have two choices to fix the array.
            // Either modify the current element to be equal to the next element,
            // Or modify the next element to be equal to the current element.
            if (current > next) {
                // Temporarily modify the current element
                nums[i] = next;

                // Check if the array is sorted after this modification
                if (std::is_sorted(nums.begin(), nums.end())) {
                    return true; // If sorted, return true
                }

                // Undo the modification of the current element
                nums[i] = current;

                // Permanently modify the next element to match the current element
                nums[i + 1] = current;

                // Return whether the array is sorted after modifying the next element
                return std::is_sorted(nums.begin(), nums.end());
            }
        }

        // If we never found a pair that needed fixing, the array is already
        // non-decreasing, so we return true.
        return true;
    }
};
```

TypeScript

```typescript
// This function checks if the array can be made non-decreasing by modifying at most one element.
function checkPossibility(nums: number[]): boolean {
    // Helper function to determine whether the array is sorted in non-decreasing order.
    const isNonDecreasing = (arr: number[]): boolean => {
        for (let i = 0; i < arr.length - 1; ++i) {
            if (arr[i] > arr[i + 1]) {
                return false;
            }
        }
        return true;
    };

    // Main loop to check each pair of elements in the array.
    for (let i = 0; i < nums.length - 1; ++i) {
        const current = nums[i],
            next = nums[i + 1];

        // If the current element is greater than the next element.
        // we try to make an adjustment and check if it resolves the non-decreasing order issue.
        if (current > next) {
            const temp = nums[i]; // Keep the original value to restore later if needed.

            nums[i] = next; // Try lowering the current value to the next one's value.
            if (isNonDecreasing(nums)) {
                return true;
            }

            nums[i] = temp; // Restore the original value as the lowering approach did not work.
            nums[i + 1] = current; // Try raising the next value to the current one's value.

            // Return whether this adjustment resulted in a non-decreasing array.
            return isNonDecreasing(nums);
        }
    }

    // If no adjustments were needed, the array is already non-decreasing.
    return true;
}
```

```python
from typing import List


class Solution:
    def check_possibility(self, nums: List[int]) -> bool:
        # Helper function to check if the list is non-decreasing.
        def is_sorted(arr: List[int]) -> bool:
            for i in range(len(arr) - 1):
                if arr[i] > arr[i+1]:
                    return False
            return True

        n = len(nums)
        # Go through each pair in the list.
        for i in range(n - 1):
            # If the current element is greater than the next element, a modification is needed.
            if nums[i] > nums[i + 1]:
                # Temporarily change the current element to the next one and check if sorted.
                temp = nums[i]
                nums[i] = nums[i + 1]
                if is_sorted(nums):
                    return True
                # If not sorted, revert the change and modify the next element instead.
                nums[i] = temp
                nums[i + 1] = temp
                return is_sorted(nums)
        # If no modification needed, then it is already non-decreasing.
        return True


# Example usage:
# sol = Solution()
# print(sol.check_possibility([4,2,3])) # True
# print(sol.check_possibility([4,2,1])) # False
```

Time and Space Complexity

Time Complexity

The time complexity of the `checkPossibility` function consists of a `for` loop that iterates through the elements of `nums` once, and a potential two calls to the `is_sorted` helper function within the loop if a disorder is found. The `is_sorted` function iterates through the elements of `nums` once to compare adjacent pairs.

Let's break it down:

- The `for` loop runs in `O(n)` where `n` is the number of elements in `nums`.
- The `is_sorted` function is `O(n)`, as it evaluates all adjacent pairs in `nums`.
- In the worst-case scenario, `is_sorted` is called twice, maintaining `O(n)` for each call.

Thus, the time complexity is `O(n)` for the single loop plus `O(n)` for each of the two possible calls to `is_sorted`, which yields a worst-case time complexity of `O(n + 2n)`, simplified to `O(n)`.

Space Complexity

The space complexity of the function is primarily `O(1)` since the function only uses a constant extra space for the variables `a`, `b`, and `i`, and modifies the input list `nums` in-place.

However, considering the creation of the iterator over the pairwise comparison, if it is not optimized away by the interpreter, may in some cases add a small overhead, but this does not depend on the size of the input and hence remains `O(1)`.