1752. Check if Array Is Sorted and Rotated

Easy Array

Problem Description

The problem requires us to determine if a given array `nums` can be the result of rotating a sorted array. A sorted array is considered non-decreasing, meaning that the elements are in ascending order or equal. The array may contain duplicate elements.

A rotation means taking any number of elements from one end of the array and moving them to the other end without changing their order. For example, [3, 4, 5, 1, 2] is a rotated version of [1, 2, 3, 4, 5]. The challenge is to figure out if the `nums` array could be made from such a rotation of a sorted array. If the array has not been rotated or has been rotated in a way that it still maintains non-decreasing order, our function should return `true`; otherwise, it should return `false`.

One important property of a rotated non-decreasing array is that it will have at most one point where the next number is less than the current number, which happens at the rotation point. In a purely sorted array, this does not happen at all.

Intuition

The intuition behind the solution approach is the property of the rotated sorted array mentioned earlier: if we look at each pair of consecutive elements in the array, going from the start to the end, there should be at most one occurrence where a number is greater than the number that follows it.

The method `check` in the provided Python code implements this logic. It uses list comprehension to compare each pair of elements in the array (handling the edge case of the first element by using modulo indexing), and counts how many times `nums[i - 1] > nums[i]` occurs. This would equal zero for an unrotated sorted array and exactly one for a correctly rotated sorted array. If this count is greater than one, then there must be at least two decreases, indicating that the array is neither sorted nor a rotated version of a sorted array, so the function will return `false`. Otherwise, it returns `true`.

Solution Approach

The solution implements a simple but clever algorithm that leverages the nature of a rotated, sorted list to count how many times an element is followed by a smaller element. The crux is that if a list is sorted and then rotated, it will contain at most one such occurrence (the rotation pivot), and if that list had been rotated more than once, there should be two or more such occurrences.

No specific data structures are required for this approach as it works directly on the given list of numbers. The algorithm follows these steps:

1. Initialize a counter to zero. This will count the occurrences where a number is followed by a smaller number.
2. Iterate through the list of numbers using Python's `enumerate` function, which gives both the index `i` and the value `v` at each step.
3. Compare each number with the next one (but since it could wrap around, the previous to the first is the last, hence `nums[i - 1] > v`). If the previous number is greater, we have found an occurrence and the list comprehension turns it into a `1`, otherwise a `0`.
4. Sum the values from the list comprehension. This sum represents the number of decreases found in the list.
5. If the sum is greater than `1`, return `false`, since this indicates the list was not just rotated from a sorted list (it was either unsorted before or tampered with after rotation).
6. Otherwise, return `true`, which implies it is either the original sorted list or a properly rotated version.

Here's how the code works:

```
class Solution:
    def check(self, nums: List[int]) -> bool:
        # Use list comprehension to sum the occurrences of `nums[i-1] > nums[i]`
        # Enclose this condition in parentheses to get a boolean (True/False) value which translates to (1/0)
        # when summed. This conditional will be True at most once in a rotated array.
        # Use `enumerate` to get both index and value for each element
        # Return True if there are 0 or 1 occurrences of a drop from `nums[i-1]` to `nums[i]`, otherwise False
        return sum(nums[i - 1] > v for i, v in enumerate(nums)) <= 1
```

The algorithm's complexity is O(n), where n is the number of elements in the list, because it involves one pass over the data. No sorting or nested loops are involved, making it efficient for this use case.

Example Walkthrough

Let's assume we have the following array: `[4, 5, 1, 2, 3]`. According to our problem description, we need to determine if this could be the result of a sorted array that has been rotated.

Following our solution approach:

1. We initialize our counter to zero. This will keep track of the number of times an element is followed by a smaller element.
2. We iterate through the array `[4, 5, 1, 2, 3]` using Python's `enumerate` function:
   - Comparing 4 (index 0) with 5 (index 1) yields `False` (since 4 < 5, there is no decrease here).
   - Comparing 5 (index 1) with 1 (index 2) yields `True` (5 > 1, so we have a decrease).
   - Comparing 1 (index 2) with 2 (index 3) yields `False`.
   - Comparing 2 (index 3) with 3 (index 4) yields `False`.
   - Lastly, since the array might be rotated, we also compare the last and the first element: 3 (last) with 4 (first) yields `False`.
3. We count the number of `True` values from these comparisons using a list comprehension.
   - We have one `True` in our example which means the count is 1.
4. The sum of decreases is 1, which is not greater than 1, so according to our rule:
5. We return `true`, indicating that our array `[4, 5, 1, 2, 3]` could have been a sorted array that was rotated.

This example illustrates a case where the array is a rotated version of a sorted array as there is only one point of decrease, which aligns with the 'rotated sorted array' property.

Solution Implementation

Python

```
from typing import List

class Solution:
    def check(self, nums: List[int]) -> bool:
        # Initialize count to track the number of times the
        # current number is less than the previous number in the list
        decrease_count = 0

        # Iterate over the list of numbers along with the index
        for i, value in enumerate(nums):
            # If the current value is less than the previous value (circularly),
            # we increment the decrease_count.
            # nums[i - 1] accesses the previous element since Python supports
            # negative indexing, for the first element it will compare with the last element
            if nums[i - 1] > value:
                decrease_count += 1

        # The array is considered sorted and rotated at most once if there's zero or one decrease
        return decrease_count <= 1
```

Java

```
class Solution {
    // Method to check if the array can be non-decreasing by modifying at most one element.
    public boolean check(int[] nums) {
        // Variable to keep track of the number of times a pair is out of order.
        int countOutOfOrder = 0;

        // 'n' is the length of the array.
        int n = nums.length;

        // Iterate over the elements of the array.
        for (int i = 0; i < n; ++i) {
            // Check if the current element is greater than the next element
            // The next element of the last item is the first item, hence the modulo operation.
            if (nums[i] > nums[(i + 1) % n]) {
                // Increment the out of order count.
                ++countOutOfOrder;
            }
        }

        // The array is non-decreasing if there is at most one out-of-order pair.
        return countOutOfOrder <= 1;
    }
}
```

C++

```
class Solution {
public:
    // Function to check if the array is non-decreasing
    // by making at most one modification (which is essentially
    // the same as checking whether the array is a rotated
    // version of a non-decreasing array).
    bool check(vector<int>& nums) {
        // Initialize the count of "out-of-order" pairs to 0.
        int outOfOrderCount = 0;

        // Get the size of the array.
        int arraySize = nums.size();

        // Loop through the array elements.
        for (int i = 0; i < arraySize; ++i) {
            // Increase the "out-of-order" count if the current element
            // is greater than the next element. We use modulo operation
            // to compare the last element with the first element.
            if (nums[i] > nums[(i + 1) % arraySize]) {
                outOfOrderCount++;
            }
        }

        // The array can be considered non-decreasing if there's at most one
        // "out-of-order" pair which could be the point of rotation.
        return outOfOrderCount <= 1;
    }
};
```

TypeScript

```
/**
 * Determines if the array can be made non-decreasing by modifying at most one element.
 * A non-decreasing array is one where each element is less than or equal to the next.
 * @param {number[]} numbers - The array of numbers to check.
 * @returns {boolean} - Returns true if the array can be non-decreasing after at most one modification, otherwise false.
 */
function check(numbers: number[]): boolean {
    // Get the length of the numbers array.
    const length = numbers.length;

    // The reduce method applies a function against an accumulator and each value of the array
    // (from left-to-right) to reduce it to a single value.
    // Here, it counts the number of times an element is greater than its successor,
    // considering the array in a circular manner by using modulo operation.
    const irregularities = numbers.reduce((irregularityCount, currentValue, index) => {
        // Compare current element with the next element (wrap around using the modulo operator).
        const isIrregular = currentValue > numbers[(index + 1) % length];

        // Increment count if there's an irregularity.
        return irregularityCount + (isIrregular ? 1 : 0);
    }, 0);

    // The array can be non-decreasing if there's at most one irregularity.
    return irregularities <= 1;
}
```

```
from typing import List

class Solution:
    def check(self, nums: List[int]) -> bool:
        # Initialize count to track the number of times the
        # current number is less than the previous number in the list
        decrease_count = 0

        # Iterate over the list of numbers along with the index
        for i, value in enumerate(nums):
            # If the current value is less than the previous value (circularly),
            # we increment the decrease_count.
            # nums[i - 1] accesses the previous element since Python supports
            # negative indexing, for the first element it will compare with the last element
            if nums[i - 1] > value:
                decrease_count += 1

        # The array is considered sorted and rotated at most once if there's zero or one decrease
        return decrease_count <= 1
```

Time and Space Complexity

The time complexity of the given code is `O(n)` where `n` is the length of the input array `nums`. This is because the code iterates through the list exactly once with the `enumerate(nums)` function and performs a constant-time operation of comparison and summation in each iteration.

The space complexity of the code is `O(1)` since it only uses a fixed amount of extra space. The extra space is independent of the input size, it only includes a few variables to keep track of the count of rotations which does not scale with the size of the input list.