896. Monotonic Array

Easy Array

Leetcode Link

Problem Description

In this problem, we are given an integer array `nums` and our task is to determine whether the array is monotonic or not. An array is considered monotonic if it is either monotonically increasing or monotonically decreasing. This means that for each pair of indices `i` and `j` where `i <= j`, the condition `nums[i] <= nums[j]` must hold true for the entire array if it's increasing, or `nums[i] >= nums[j]` if it's decreasing. The goal is to return `true` if the array meets either of these conditions for all its elements, and `false` otherwise.

Intuition

To understand if an array is monotonic, we need to check two conditions:

1. Monotone Increasing: To verify this, we need to check if every element at index `i` is less than or equal to the element at index `i+1`. This should be true for all consecutive pairs in the array.
2. Monotone Decreasing: Similarly, we need to check if every element at index `i` is greater than or equal to the element at index `i+1` for all consecutive pairs.

The given Python solution approaches the problem by checking both conditions separately. It uses the `pairwise` utility which, in this context, would generate pairs of consecutive elements from the `nums` array. It is important to note that `pairwise` is available in the `itertools` module from Python 3.10 onwards. For versions prior to that, we can manually create pairs using a simple loop or list comprehension.

For the increasing condition, the expression `all(a <= b for a, b in pairwise(nums))` returns `True` if every element `a` is less than or equal to the next element `b`, for all pairs `(a, b)` in the array. Similarly, the decreasing condition is checked with the expression `all(a >= b for a, b in pairwise(nums))`.

Finally, the function returns `True` if either of these conditions is `True`, meaning the array is either strictly non-decreasing or strictly non-increasing. Otherwise, it returns `False`, indicating the array is not monotonic.

Solution Approach

The solution provided in the Python code relies on a straightforward approach and the efficient use of Python's built-in functions to check if the array is monotonic. Let's break down the steps of how the algorithm is implemented:

1. The solution first attempts to determine if the array is monotonically increasing. It does this with the help of a generator expression `all(a <= b for a, b in pairwise(nums))`. This expression generates a sequence of boolean values for each pairwise comparison between items `a` and `b` in the array such that `a` and `b` are consecutive elements. If `a` is always less than or equal to `b`, meaning no violation of the increasing condition is found, the `all` function will return `True`.

2. Similarly, the solution tries to find out if the array is monotonically decreasing by using the expression `all(a >= b for a, b in pairwise(nums))`. This also generates a sequence of boolean values where each pair is compared to ensure that `a` is greater than or equal to `b`. If this is true for the whole array, the `all` function returns `True`.

3. As for the `pairwise` function, it is not explicitly defined within the solution, which implies it must be imported from Python's `itertools` module before using the solution. `pairwise` creates an iterator that returns consecutive pairs of elements from the input iterable. For example, `pairwise([1, 2, 3, 4])` would yield (1, 2), (2, 3), and (3, 4). If the `pairwise` function is unavailable, the same effect can be achieved with `zip(nums, nums[1:])`.

4. The crucial part of the solution is the `return incr or decr` line of code. What it does is return `True` if either variable `incr` or `decr` is `True`. These two variables hold the outcomes of the monotonic increasing or decreasing checks. In essence, the array is monotonic if it is either entirely non-decreasing or non-increasing.

By combining these checks, the problem is addressed in a concise manner that effectively determines the monotonicity of the array with minimal iteration, therefore optimizing the solution's time complexity to O(n), where n is the length of the input array.

Example Walkthrough

Let's illustrate the solution approach using a small example array `nums = [3, 3, 5, 5, 6, 7, 7]`.

1. First, we create pairs of consecutive elements:
   - (3, 3)
   - (3, 5)
   - (5, 5)
   - (5, 6)
   - (6, 7)
   - (7, 7)
2. Now, we apply the check to see if the array is monotonically increasing. For this example:
   - We compare `3 <= 3`, which is `True`.
   - We compare `3 <= 5`, which is `True`.
   - We compare `5 <= 5`, which is `True`.
   - We compare `5 <= 6`, which is `True`.
   - We compare `6 <= 7`, which is `True`.
   - We compare `7 <= 7`, which is `True`.

   These pairwise comparisons give us all `True` outcomes. Therefore, `incr = True` as all elements satisfy the condition `a <= b`.
3. We don't necessarily need to proceed with checking for monotonically decreasing conditions because we have already found that the array is monotonically increasing. However, for the sake of understanding, we would check as follows:
   - Compare `3 >= 3`, which is `True`.
   - Compare `3 >= 5`, which is `False`.

   At this point, we already have a `False`, so we know `decr` will be `False`, and there's no need to continue. Every subsequent comparison (though not needed here) would have at least one more `False`, confirming the array isn't monotonically decreasing.
4. Since the variable `incr` holds `True` (and `decr` holds `False`), the final result returned by the function is `True`.

In this example, we deduced that `nums` is monotonically increasing, and therefore, it is a monotonic array. The key takeaway is that the array only needs to fulfill one of the two monotonic conditions (increasing or decreasing), not both.

Python Solution

```python
from itertools import pairwise  # Ensure that pairwise is imported from itertools

class Solution:
    def isMonotonic(self, numbers: List[int]) -> bool:
        # Check if the sequence is monotonically increasing
        is_increasing = all(a <= b for a, b in pairwise(numbers))
        # Check if the sequence is monotonically decreasing
        is_decreasing = all(a >= b for a, b in pairwise(numbers))
        # The sequence is monotonic if it's either increasing or decreasing
        return is_increasing or is_decreasing

# Note: If the Python environment is older than 3.10, you'll need this definition of pairwise:
# def pairwise(iterable):
#     "s -> (s0,s1), (s1,s2), (s2, s3), ..."
#     a, b = tee(iterable)
#     next(b, None)
#     return zip(a, b)

```

Java Solution

```java
class Solution {
    // Function to determine if the array is either entirely non-increasing or non-decreasing
    public boolean isMonotonic(int[] nums) {
        boolean isIncreasing = true; // To keep track if the array is non-decreasing
        boolean isDecreasing = true; // To keep track if the array is non-increasing

        // Iterate over the array starting from the second element
        for (int i = 1; i < nums.length; i++) {
            if (nums[i] < nums[i - 1]) {
                // If the current number is less than the previous, it's not non-decreasing
                isIncreasing = false;
            }
            if (nums[i] > nums[i - 1]) {
                // If the current number is greater than the previous, it's not non-increasing
                isDecreasing = false;
            }
            // If the array is neither non-decreasing nor non-increasing, return false
            if (!isIncreasing && !isDecreasing) {
                return false;
            }
        }
        // If we reach this point, the array is either non-decreasing, non-increasing, or all elements are equal
        return true;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to determine if the array is monotonic (either entirely non-increasing or non-decreasing)
    bool isMonotonic(vector<int>& nums) {

        // Initialize two boolean flags for increasing and decreasing
        bool isIncreasing = true;
        bool isDecreasing = true;

        // Iterate over the array starting from the second element
        for (int i = 1; i < nums.size(); ++i) {

            // If the current element is smaller than the previous, it's not increasing
            if (nums[i] < nums[i - 1]) {
                isIncreasing = false;
            }

            // If the current element is larger than the previous, it's not decreasing
            if (nums[i] > nums[i - 1]) {
                isDecreasing = false;
            }

            // If it's neither increasing nor decreasing, it's not monotonic
            if (!isIncreasing && !isDecreasing) {
                return false;
            }
        }

        // If the array is either increasing or decreasing, then it's monotonic
        return true;
    }
};

```

Typescript Solution

```typescript
/**
 * Determines if an array of numbers is monotonic.
 * An array is monotonic if it is either monotone increasing or monotone decreasing.
 * @param {number[]} nums The array of numbers to check.
 * @returns {boolean} True if the array is monotonic, otherwise false.
 */
function isMonotonic(nums: number[]): boolean {
    const length = nums.length;
    let isIncreasing = false;
    let isDecreasing = false;

    // Traverse the array, starting from the second element
    for (let i = 1; i < length; i++) {
        const previous = nums[i - 1]; // Previous element
        const current = nums[i]; // Current element

        // Check if the current pair is increasing
        if (previous < current) {
            isIncreasing = true;
        }
        // Check if the current pair is decreasing
        else if (previous > current) {
            isDecreasing = true;
        }

        // If the sequence has both increasing and decreasing pairs,
        // it is not monotonic.
        if (isIncreasing && isDecreasing) {
            return false;
        }
    }

    // If the loop completes without returning false,
    // the array is monotonic.
    return true;
}

```

Time and Space Complexity

The time complexity of the code is `O(n)` where `n` is the length of the `nums` list. This is because the `pairwise` function is going through the list only once for each check (increasing and decreasing). Each `all()` call iterates over the list in a pairwise fashion, which means there will be a total of `n-1` comparisons for each call.

The space complexity of the code is `O(1)` since the space used does not depend on the size of the input `nums` list. The `pairwise` function generates a sequence of tuples which is consumed by the `all()` function, and this does not require additional space that scales with the input size.