775. Global and Local Inversions

Medium Array Math

Leetcode Link

Problem Description

You are provided with an integer array `nums` which is a permutation of all integers in the range `[0, n - 1]`, where `n` is the length of `nums`. The concept of inversions in this array is split into two types: **global inversions** and **local inversions**.

- **Global inversions:** These are the pairs `(i, j)` such that `i < j` and `nums[i] > nums[j]`. Essentially, a global inversion is any two elements that are out of order in the entire array.
- **Local inversions:** These are the specific cases where `nums[i] > nums[i + 1]`. This means that each local inversion is a global inversion where the elements are adjacent to each other.

Your task is to determine whether the number of **global inversions** is exactly the same as the number of **local inversions** in the array. If they are equal, return `true`, otherwise return `false`.

Intuition

To solve this problem, we need to understand that all local inversions are also global inversions by definition, since adjacent out-of-order elements also count as a pair of out-of-order elements in the greater array.

However, not all global inversions are local; there can be non-adjacent elements that are out of order. For the numbers of local and global inversions to be equal, there must not be any global inversions that are not also local.

This constraint means that any element `nums[i]` must not be greater than `nums[j]` for `j > i + 1`. So instead of counting inversions which would take O(n^2) time, we can simply look for the presence of any such global inversion that is not local.

Given this understanding, the solution avoids a brute-force approach and cleverly checks for the condition that forbids equal global and local inversions. As we iterate through the array starting from the third element (`index 2`), we keep track of the maximum number we've seen up to two positions before the current index.

If at any point this maximum number is greater than the current number, `mx > nums[i]`, then a non-local global inversion is found, and we immediately return `false`.

If we finish the loop without finding any non-local global inversions, then all global inversions must also be local inversions, and we return `true`.

Solution Approach

The Reference Solution Approach contains a thoughtfully written `isIdealPermutation` method which leverages the insight that in a permutation array where the number of global inversions is to be equal to the local inversions, no element can be out of order by more than one position. This is because any such displacement would constitute a global inversion that is not a local inversion, thereby invalidating our condition for equality between the two types of inversions.

Python Code Walkthrough

In the provided Python solution, we see a single `for` loop that starts at the third element of the array (`i = 2`), and at each step of the iteration, the loop does the following:

1. Update the `mx` variable to hold the maximum value found so far in `nums`, but strictly considering elements up to two places before the current index `i`. This is accomplished with the expression `(mx := max(mx, nums[i - 2]))`, which is using the walrus operator (`:=`) introduced in Python 3.8. This operator allows variable assignment within expressions.
2. It then compares the maximum value `mx` found within the previous two elements with the current element `nums[i]`. If `mx` is found to be greater than `nums[i]`, this indicates the presence of a non-local global inversion, and the function returns `False` immediately.
3. If the loop completes without finding any such condition, it implies that there are no non-local global inversions and therefore all global inversions are indeed local. Hence, the function returns `True`.

This approach is efficient because it runs in O(n) time, where `n` is the length of `nums`. The use of the maximum value `mx` and the iteration from the third element is critical because it leverages the rule that elements cannot be out of place by more than one position for a permutation to have equal numbers of local and global inversions. This is a great example of how understanding the fundamental properties of a problem can lead to elegant and efficient solutions.

Example Walkthrough

Let's consider a small example to illustrate the solution approach using the array `nums = [1, 0, 3, 2]`.

In this array:

- The array has length `n = 4`.
- The array is a permutation of integers `[0, 1, 2, 3]`.

Following the solution approach:

1. We start by initializing a variable `mx` which holds the maximum value up to two elements before the current index. Initially, `mx` does not have a value as we start checking from the third element.
2. We now iterate through the array starting from index `i = 2`.
   - At `i = 2`, our current element is `nums[2] = 3`. The maximum value up to two elements before index `2` is `max(nums[0], nums[1]) = max(1, 0) = 1`. Since `mx = 1` is not greater than `nums[2] = 3`, we continue to the next element.
   - At `i = 3`, our current element is `nums[3] = 2`. We update `mx` to the maximum value found in the window up to two indices before `i = 3`, which is now `mx = max(mx, nums[1]) = max(1, 0) = 1`. We compare `mx` with `nums[3]`. Here, `mx = 1` is not greater than `nums[3] = 2`, so we continue.
3. Since we did not find any case where `mx > nums[i]`, we have confirmed that there are no global inversions that are not local. Therefore, our function `isIdealPermutation` will return `True` for this array.

This simple example confirms that for this particular permutation of `nums`, the number of global inversions is exactly the same as the number of local inversions, adhering to the solution approach described. The function correctly identifies this by checking if any element is displaced by more than one position from its original location, which in this case, it is not.

Python Solution

```python
from typing import List

class Solution:
    def isIdealPermutation(self, nums: List[int]) -> bool:
        # Initialize a variable to keep track of the maximum number seen so far.
        # Start with the first element as we will begin checking from the third element.
        max_seen = 0

        # Iterate over the array starting from the third element (index 2)
        for i in range(2, len(nums)):
            # Update the max_seen with the largest value among itself and
            # the element two positions before the current one.
            max_seen = max(max_seen, nums[i - 2])

            # If the max_seen so far is greater than the current element,
            # it is not an ideal permutation, so return False.
            if max_seen > nums[i]:
                return False

        # If the loop completes without returning False,
        # all local inversions are also global inversions, hence it's an ideal permutation.
        return True

# Example usage:
# sol = Solution()
# print(sol.isIdealPermutation([1, 0, 2])) # Should return True

```

Java Solution

```java
class Solution {

    // This method checks if the number of global inversions is equal to the number of local inversions
    // in the array, which is a condition for the array to be considered an ideal permutation.
    public boolean isIdealPermutation(int[] nums) {
        // Initialize the maximum value found to the left of the current position by two places.
        // We start checking from the third element (at index 2), since we are interested in comparing
        // it with the value at index 0 for any inversion that isn't local.
        int maxToLeftByTwo = 0;

        // Loop through the array starting from the third element.
        // We don't need to check the first two elements because any inversion there is guaranteed to be local.
        for (int i = 2; i < nums.length; ++i) {
            // Update maxToLeftByTwo to the highest value found so far in nums,
            // considering elements two positions to the left of the current index.
            maxToLeftByTwo = Math.max(maxToLeftByTwo, nums[i - 2]);

            // If the maximum value to the left (by two positions) is greater than the current element,
            // it means there's a global inversion, and the array cannot be an ideal permutation.
            if (maxToLeftByTwo > nums[i]) {
                return false;
            }
        }

        // If the loop completes without finding any global inversions other than local ones,
        // the array is an ideal permutation.
        return true;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // Include necessary headers

class Solution {
public:
    // Check if the given permutation is an ideal permutation
    bool isIdealPermutation(vector<int>& nums) {
        // Initialize the maximum value found so far to the smallest possible integer
        int maxVal = 0;

        // Start iterating from the third element in the array
        for (int i = 2; i < nums.size(); ++i) {
            // Update the maximum value observed in the prefix of the array (till nums[i-2])
            maxVal = max(maxVal, nums[i - 2]);

            // If at any point the current maximum is greater than the current element,
            // we don't have an ideal permutation, so return false
            if (maxVal > nums[i]) return false;
        }

        // If the loop completes without returning false, it's an ideal permutation
        return true;
    }
};

```

Typescript Solution

```typescript
// Import necessary functions from standard modules
import { max } from 'lodash';

// Check if the given permutation is an ideal permutation
function isIdealPermutation(nums: number[]): boolean {
    // Initialize the maximum value found so far to the first element
    // or to the smallest possible integer if the array is empty.
    let maxValue: number = nums.length > 0 ? nums[0] : Number.MIN_SAFE_INTEGER;

    // Start iterating from the third element in the array
    for (let i: number = 2; i < nums.length; i++) {
        // Update the maximum value observed in the prefix of the array (up to nums[i - 2])
        maxValue = max([maxValue, nums[i - 2]])!;

        // If at any point the current maximum is greater than the current element,
        // we don't have an ideal permutation, so return false
        if (maxValue > nums[i]) {
            return false;
        }
    }

    // If the loop completes without returning false, it's an ideal permutation
    return true;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(n)`, where `n` is the length of the input list `nums`. This is because the for loop iterates from `2` to `n`, performing a constant amount of work for each element by updating the `mx` variable with the maximum value and comparing it with the current element.

Space Complexity

The space complexity of the code is `O(1)`, which means it uses a constant amount of extra space. No additional data structures are used that grow with the input size; only the `mx` variable is used for keeping track of the maximum value seen so far, which does not depend on the size of `nums`.