2340. Minimum Adjacent Swaps to Make a Valid Array

Medium Greedy Array   Leetcode Link

Problem Description

You are provided with an array `nums` which is zero-indexed, meaning the indexing starts from 0. Your task is to sort the array in a specific way using only swaps between two adjacent elements. The goal is to have the smallest number positioned at the start (leftmost) of the array and the largest number at the end (rightmost) of the array. The array is considered **valid** when these conditions are met. Your objective is to determine the minimum number of such swaps required to make the array valid.

Intuition

To solve this problem, the strategy is to find the positions of the smallest and the largest elements in the array since only their positions matter for making the array valid. Typically, a linear scan through the array allows us to identify these elements and their indices.

Once the positions of the smallest and largest elements are known, there are a few cases to consider:

- If the smallest element is already at the first (leftmost) position and the largest element is already at the last (rightmost) position, then no swaps are needed.
- If the smallest and largest elements are not in their correct positions, they need to be swapped towards their respective ends. However, if the largest element is to the left of the smallest element, when the largest element is moved to the end, it effectively takes one swap less since the smallest element moves one place towards the beginning in the process.

The formula `i + len(nums) - 1 - j - (i > j)` reflects this logic, where `i` is the index of the smallest element, `j` is the index of the largest element, and `len(nums)` is the size of the array. The term `(i > j)` is a conditional that subtracts one from the total swap count if the largest element comes before the smallest element.

Solution Approach

The solution provided uses a single-pass algorithm to find the indices `i` and `j`, which represent the positions of the smallest and the largest elements in the `nums` array respectively. The algorithm iterates over each element in the array using a `for` loop and checks if the current element is less than or equal to the smallest element found so far, or if it is greater than or equal to the largest element found so far.

Here is the breakdown of how it works:

1. Two variables `i` and `j` are initialized to 0, indicating that initially, we consider the first element as both the smallest and the largest.

2. A `for` loop begins which examines each element indexed by `k` and compares its value `v` with the current smallest and the largest elements.

3. For the smallest element (`nums[i]`), we have two conditions:

   - If `v` is less than `nums[i]`, we update `i` to `k` because we have found a new smallest element.

   - If `v` is equal to `nums[i]` and `k` is less than `i`, we update `i` to `k` because we want the smallest element that is closest to the start of the array.

4. Similarly, for the largest element (`nums[j]`), we have two conditions:

   - If `v` is greater than `nums[j]`, we update `j` to `k` because we found a new largest element.

   - If `v` is equal to `nums[j]` and `k` is greater than `j`, we update `j` to `k` because we want the largest element that is closest to the end of the array.

5. After the loop ends, we have the positions of the smallest and largest elements in `i` and `j`. The number of swaps required is then calculated with the following formula:

   ```
   i + len(nums) - 1 - j - (i > j)
   ```

   The logic behind this formula is:

   - `i + len(nums) - 1 - j` calculates the total distance the smallest and largest elements need to move to reach their respective ends.

   - `(i > j)` serves as an adjustment in case the largest element is before the smallest element. If `i` is greater than `j`, we have moved the smallest element one step to the left already when moving the largest element to the end, thus requiring one less swap.

6. The code finally returns `0` if `i` is equal to `j`, which implies that the smallest and largest element is the same, hence no swaps required. Otherwise, it returns the calculated number of swaps needed to organize the array properly.

This concise algorithm effectively solves the problem with a time complexity of `O(n)` since it requires only one pass through the array, and a space complexity of `O(1)` as it uses a constant amount of space.

Example Walkthrough

Let's go through a small example to illustrate the solution approach. Consider the array `nums = [3, 1, 2, 4]`.

1. Initialize `i` and `j` to `0`. Initially, we consider the first element as the smallest and the largest one. So `i = j = 0`.

2. Start the `for` loop with index `k`, iterating through the array from `k = 1` to the end. The comparison process is as follows:

   - For `k = 1`: `v = nums[1] = 1`, which is less than `nums[i] = 3`. Thus, `i` is updated to `1`.

   - For `k = 2`: `v = nums[2] = 2`. It doesn't change `i` because `v` is greater than `nums[i] = 1`, and it doesn't change `j` because `v` is less than `nums[j] = 3`.

   - For `k = 3`: `v = nums[3] = 4`, which is greater than `nums[j] = 3`. Thus, `j` is updated to `3`.

3. Now we have the positions of the smallest (`1` for `nums[i]`) and largest elements (`3` for `nums[j]`). We use the swap calculation formula:

   ```
   i + len(nums) - 1 - j - (i > j)
   ```

   Here, `len(nums)` is `4`, so plugging in the values, we get:

   ```
   1 + 4 - 1 - 3 - (1 > 3) => 1 + 4 - 1 - 3 - 0 => 1
   ```

   Therefore, the minimum number of swaps required is `1`.

4. The array after the swap process will look like this: `[1, 3, 2, 4]`, where the smallest number `1` has been brought to the start and the largest `4` is at the end.

This walk-through demonstrates that a single scan of the array is sufficient to find the required number of swaps to sort the array according to the given conditions, and the calculation is straightforward once we have the positions of the smallest and largest elements.

Python Solution

```python
from typing import List

class Solution:
    def minimumSwaps(self, nums: List[int]) -> int:
        # Initialize the positions of the minimum and maximum elements
        min_position = max_position = 0

        # Iterate through the array to find the positions
        for index, value in enumerate(nums):
            # Update the position of the minimum element if a
            # new minimum is found or if the same minimum is found at a lower index
            if value < nums[min_position] or (value == nums[min_position] and index < min_position):
                min_position = index

            # Update the position of the maximum element if a
            # new maximum is found or if the same maximum is found at a higher index
            if value > nums[max_position] or (value == nums[max_position] and index > max_position):
                max_position = index

        # Calculate the number of swaps needed
        swaps = 0 if min_position == max_position else min_position + (len(nums) - 1 - max_position)

        # If min_position is greater than max_position, one swap has been double counted
        if min_position > max_position:
            swaps -= 1

        return swaps

```

Java Solution

```java
class Solution {
    // Function to find the minimum number of swaps required to make the given array sorted
    public int minimumSwaps(int[] nums) {
        int n = nums.length; // Length of the given array
        int minIndex = 0, maxIndex = 0; // Initialize indices for minimum and maximum elements

        // Loop through the array to find the indices for the minimum and maximum elements
        for (int k = 0; k < n; ++k) {
            // Update the index of the minimum element found so far
            if (nums[k] < nums[minIndex] || (nums[k] == nums[minIndex] && k < minIndex)) {
                minIndex = k;
            }
            // Update the index of the maximum element found so far
            if (nums[k] > nums[maxIndex] || (nums[k] == nums[maxIndex] && k > maxIndex)) {
                maxIndex = k;
            }
        }

        // If the minimum and maximum elements are at the same position, no swaps are needed
        if (minIndex == maxIndex) {
            return 0;
        }

        // Calculate the number of swaps required
        // The calculation is done by considering the positions of the minimum and maximum elements
        // and adjusting the swap count depending on their relative positions
        int swaps = minIndex + n - 1 - maxIndex - (minIndex > maxIndex ? 1 : 0);

        // Return the number of swaps
        return swaps;
    }
}

```

C++ Solution

```cpp
#include <vector>
using namespace std;

class Solution {
public:
    int minimumSwaps(vector<int>& nums) {
        int numsSize = nums.size();         // Get the size of the input array
        int minIndex = 0, maxIndex = 0;     // Initialize the indices for the minimum and maximum elements

        // Loop through the array to find the minimum and maximum elements' indices
        for (int k = 0; k < numsSize; ++k) {
            // Update the index of the minimum element if a smaller element is found
            // or if the same element is found at a smaller index
            if (nums[k] < nums[minIndex] || (nums[k] == nums[minIndex] && k < minIndex)) {
                minIndex = k;
            }
            // Update the index of the maximum element if a larger element is found
            // or if the same element is found at a larger index
            if (nums[k] > nums[maxIndex] || (nums[k] == nums[maxIndex] && k > maxIndex)) {
                maxIndex = k;
            }
        }

        // If the minimum and maximum elements are at the same index, no swaps are needed
        if (minIndex == maxIndex) {
            return 0;
        }

        // Calculate the number of swaps needed
        // If minIndex is greater than maxIndex, one swap will be counted twice, so subtract one
        int swaps = minIndex + numsSize - 1 - maxIndex - (minIndex > maxIndex);
        return swaps;
    }
};

```

Typescript Solution

```typescript
/**
 * Calculates the minimum number of swaps needed to bring the minimum and maximum elements
 * to the ends of the array, with the minimum element at the start and the maximum at the end.
 * @param {number[]} nums - Array of numbers for which to calculate the minimum number of swaps.
 * @returns {number} - The minimum number of swaps required.
 */
function minimumSwaps(nums: number[]): number {
    let minIndex = 0; // Index for the minimum element
    let maxIndex = 0; // Index for the maximum element
    const length = nums.length; // The length of the nums array

    // Iterate over the array to find the indices of the minimum and maximum elements.
    for (let k = 0; k < length; ++k) {
        // Update minIndex if a smaller element is found or if the same element is found at a lower index
        if (nums[k] < nums[minIndex] || (nums[k] === nums[minIndex] && k < minIndex)) {
            minIndex = k;
        }
        // Update maxIndex if a larger element is found or if the same element is found at a higher index
        if (nums[k] > nums[maxIndex] || (nums[k] === nums[maxIndex] && k > maxIndex)) {
            maxIndex = k;
        }
    }

    // Calculate the number of swaps required.
    return minIndex === maxIndex
        ? 0 // If minIndex and maxIndex are the same, no swaps are needed.
        : minIndex + (length - 1 - maxIndex) - (minIndex > maxIndex ? 1 : 0);
    // The total swaps are normally the distance of minIndex from the start plus
    // distance of maxIndex from the end. If minIndex is after maxIndex, reduce one swap.
}

```

Time and Space Complexity

Time Complexity

The time complexity of the provided code is primarily determined by the single loop that iterates through the array `nums`. Since the loop runs for each element in the array, the time complexity is `O(n)`, where `n` is the length of the array `nums`.

Space Complexity

The space complexity of the code is `O(1)` because it uses a fixed amount of extra space regardless of the size of the input array. The extra space is used for the variables `i`, `j`, `k`, and `v`, whose storage does not scale with the size of the input array.