1051. Height Checker

Easy | Array | Counting Sort | Sorting

Leetcode Link

Problem Description

In this problem, we are given two arrays; `expected`, which represents the non-decreasing order of students' heights for a school photo, and `heights`, which is the current order of students standing in a line for the photo. The goal is to find out how many students are standing in the wrong position compared to the expected order. To do this, we need to compare the elements of the two arrays and count the number of positions where the heights do not match.

The comparison is based on the index of the array, meaning we look at the height of each student at a particular position in the `heights` array and check if it matches the height at the same position in the `expected` array. The result will be the total number of students who need to move to match the expected height order.

Intuition

To find the solution, we need to follow these steps:

1. Create a sorted version of the `heights` array, which we can call `expected`. This sorted array represents the students in the correct order.
2. We then compare each element in the sorted `expected` array with the corresponding element in the original `heights` array.
3. Every time we find a mismatch, we know that the student is not in the correct order.
4. To find the total number of mismatches, we iterate through both arrays simultaneously, comparing the students' heights element by element.
5. Each mismatch contributes to the count of students who are not in their correct position.
6. Finally, the sum of all mismatches gives us the total number of students standing out of order.

The given solution uses list comprehension to compare the two arrays and count the mismatches using the `sum` function, all in one compact line of code.

Solution Approach

The solution provided implements a straightforward algorithm for solving the problem using the following steps:

1. **Sorting**: We start by sorting the `heights` array which gives us the `expected` order of heights. Sorting is done using Python's built-in `sorted()` function, which typically implements the Timsort algorithm—a hybrid sorting algorithm derived from merge sort and insertion sort. The complexity of this operation is O(n log n), where n is the number of students.

```
expected = sorted(heights)
```

2. **Comparison**: After sorting, we need to compare the elements of the `expected` (sorted) array with the `heights` (original) array. This is done to check for any discrepancies between the current order and the expected order.

3. **Count Mismatches**: We count the number of positions where the heights do not match by iterating through both arrays simultaneously. For each pair of elements at the same index, we check if they are different.

```
return sum(a != b for a, b in zip(heights, expected))
```

   - The `zip()` function is used to create pairs of elements from both arrays that share the same index.
   - The expression `(a != b for a, b in zip(heights, expected))` creates a generator that yields `True` for each mismatch and `False` for each match.
   - `sum()` is used to count how many times `True` appears in the generator, which corresponds to the number of mismatches since `True` is interpreted as `1` and `False` as `0`.
4. **Return Result**: The sum, representing the count of mismatches, is then returned as the final result.

In terms of data structures, the solution makes use of lists (arrays in other programming languages) and employs list comprehension for concise iteration and comparison. This solution is efficient due to its simplicity and Python's optimization of list operations and sorting. The space complexity of the solution is O(n), because we are creating a new list `expected` which is a sorted copy of `heights`, and the time complexity is dominated by the sorting operation, making it O(n log n).

Example Walkthrough

Let's illustrate the solution approach with a small example. Consider the following arrays:

- `heights` : [3, 1, 2, 4]
- `expected`: [1, 2, 3, 4] (this is the sorted order of `heights`)

1. **Sorting**: First, we sort the `heights` array which should look like the `expected` array:

```
expected = sorted(heights)  # expected becomes [1, 2, 3, 4]
```

2. **Comparison**: We then compare `heights` with `expected`:

Index	heights	expected	Match?
0	3	1	No
1	1	2	No
2	2	3	No
3	4	4	Yes

   We see that at indices 0, 1, and 2, the values don't match, while at index 3, they do.

3. **Count Mismatches**: We count the mismatches using the `zip()` function and a generator expression within the `sum()` function:

```
mismatches = sum(a != b for a, b in zip(heights, expected))
# The comparison (a != b for a, b in zip([3, 1, 2, 4], [1, 2, 3, 4]))
# yields [True, True, True, False] which sums up to 3.
```

4. **Return Result**: The value `3` is then returned, indicating that three students are standing in the wrong positions.

Following this example, by comparing the sorted order (which is the expected correct order) with the current heights, we identified that 3 out of the 4 students are out of order, which shows our solution correctly identifies and counts the number of mismatches.

Python Solution

```
from typing import List

class Solution:
    def heightChecker(self, heights: List[int]) -> int:
        # Create a sorted version of the heights list, which will represent
        # the expected order after sorting by height
        expected = sorted(heights)

        # Use zip() to pair elements from the actual order and the expected order
        # and then use sum() with a generator to count differences between paired elements,
        # which indicates how many students are not in the correct positions
        mismatch_count = sum(actual_height != expected_height for actual_height, expected_height in zip(heights, expected))

        # Return the total count of mismatches found
        return mismatch_count

```

Java Solution

```
class Solution {
    public int heightChecker(int[] heights) {
        // Clone the original array to create a separate array which we can sort
        int[] expectedHeights = heights.clone();

        // Sort the expectedHeights array to have the heights in ascending order
        Arrays.sort(expectedHeights);

        // Initialize a counter to keep track of the number of heights in the wrong position
        int countMismatchedHeights = 0;

        // Iterate over the heights array to compare each element with the corresponding element in the sorted expectedHeights array
        for (int i = 0; i < heights.length; ++i) {
            // If the height in original order does not match the height in sorted order, increment the counter
            if (heights[i] != expectedHeights[i]) {
                countMismatchedHeights++;
            }
        }

        // Return the total number of heights that are out of place when compared to the sorted order
        return countMismatchedHeights;
    }
}

```

C++ Solution

```
class Solution {
public:
    int heightChecker(vector<int>& heights) {
        // Create a copy of the original 'heights' vector to store the expected sorted order
        vector<int> sortedHeights = heights;

        // Sort the 'sortedHeights' vector to represent the expected heights order
        sort(sortedHeights.begin(), sortedHeights.end());

        // Initialize a counter to track the number of students not in the correct position
        int misplacedCount = 0;

        // Iterate over the original 'heights' vector to compare with the 'sortedHeights'
        for (int i = 0; i < heights.size(); ++i) {
            // Increment the counter when the current height does not match the expected height
            misplacedCount += heights[i] != sortedHeights[i];
        }

        // Return the total count of students who are not in the correct position
        return misplacedCount;
    }
};

```

Typescript Solution

```
// Define a function to count how many students are not in the correct positions based
// on their heights compared to a sorted array of heights
function heightChecker(heights: number[]): number {
    // Create a copy of the original 'heights' array to store the expected sorted order
    const sortedHeights = [...heights];

    // Sort the 'sortedHeights' array to represent the expected heights order
    sortedHeights.sort((a, b) => a - b);

    // Initialize a counter to track the number of students not in the correct position
    let misplacedCount: number = 0;

    // Iterate over the original 'heights' array to compare it with the 'sortedHeights'
    for (let i = 0; i < heights.length; i++) {
        // Increment the counter when the current height does not match the expected height
        if (heights[i] !== sortedHeights[i]) {
            misplacedCount++;
        }
    }

    // Return the total count of students who are not in the correct position
    return misplacedCount;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given function is determined by two main operations: sorting the `heights` array and comparing the elements of the two arrays.

1. Sorting: The `sorted()` function in Python uses the Timsort algorithm, which has a worst-case time complexity of `O(n log n)` where `n` is the number of elements in the `heights` array.
2. Comparison: The `sum()` function with a generator expression that iterates over the zipped pairs of the original and sorted `heights` arrays runs in `O(n)` time since each element is visited exactly once.

Therefore, the combined time complexity of these operations is `O(n log n) + O(n)` which simplifies to `O(n log n)` because the `O(n log n)` term dominates for large values of `n`.

Space Complexity

The space complexity of the function is primarily due to the storage requirement for the sorted list `expected`.

1. Creating a sorted array: This requires additional space to hold all the `n` elements from the original `heights` array, so this operation has a space complexity of `O(n)`.

The generator expression used in the sum does not require additional space as it uses the iterator protocol, therefore, the space needed for the iterators themselves is negligible.

In conclusion, the space complexity of the given function is `O(n)` due to the extra array created to hold the sorted heights.