1460. Make Two Arrays Equal by Reversing Subarrays

Easy Array Hash Table Sorting Leetcode Link

Problem Description

In this problem, you are provided with two integer arrays `target` and `arr`. Both arrays have the same number of elements. Your goal is to determine if it's possible to make the array `arr` identical to the array `target` by performing a series of operations. For each operation, you can select any non-empty subarray from `arr` and reverse it. A subarray is a contiguous part of an array. You can reverse subarrays as many times as you need. You need to return `true` if `arr` can be made equal to `target` or `false` otherwise.

Intuition

The key insight to solve this problem is to understand that reversing a subarray does not change the overall content (numbers and their frequencies) of the array; it only changes the order of elements. If two arrays contain the same elements with the same frequencies (multisets are equal), it is always possible to make one array equal to another by reversing subarrays because you can always rearrange the elements to match. Therefore, the solution does not actually involve performing the subarray reversals; instead, it involves checking whether both arrays contain the same set of elements with the same frequency.

The approach to arrive at the solution is straightforward:

1. Sort both the `target` and `arr` arrays. Sorting brings elements of the same value next to each other and thus makes it easy to compare the arrays.
2. After sorting, if both arrays are equal, it means that `arr` can be transformed into `target` through reversing subarrays. Hence, return `true`.
3. If the arrays do not match after sorting, it means `arr` cannot be made equal to `target`, so return `false`.

Solution Approach

The implementation of the solution is quite simple and uses basic algorithms and data structures. Here is a detailed explanation of the approach:

1. **Sorting Algorithm:** The core of the solution uses a sorting algorithm. Both Python's `sort()` method on lists uses TimSort, which is a hybrid sorting algorithm derived from merge sort and insertion sort. It is a stable, adaptive, and iterative merge sort that requires fewer than n log(n) comparisons when running on partially sorted arrays, which makes it very efficient.

2. **Comparison:** After sorting, the elements in both `target` and `arr` are in the same order if they are comprised of the same set of elements. The algorithm then simply compares the two sorted arrays to check for equality. This is done using the '==' operator in Python, which compares corresponding elements in both lists.

3. **Returning the Result:** If the comparison evaluates to `true`, it means that `arr` can be rearranged to match `target` by reversing subarrays; the function thus returns `true`. If the comparison is `false`, there are elements in `arr` that do not match those in `target`, indicating that no series of reversals will make the two arrays equal. In this case, the function returns `false`.

The solution does not require any additional data structures; it works with the input arrays themselves and returns a boolean value. It is also important to note that since sorting changes the original arrays, if maintaining the original order is needed for any reason, one could sort copies of the arrays instead.

Here is the final solution encapsulated in a class, as provided in the reference code:

```python
class Solution:
    def canBeEqual(self, target: List[int], arr: List[int]) -> bool:
        target.sort()
        arr.sort()
        return target == arr
```

The above solution is concise as well as efficient due to the use of sorting, which is more time and space-efficient than other methods of comparison that could involve using hash maps or multiset data structures to compare frequencies of elements.

Example Walkthrough

Let us consider a small example to illustrate the solution approach.

Suppose the `target` array is `[1, 2, 3, 4]` and the `arr` array is `[2, 4, 1, 3]`. We want to find out if we can make `arr` identical to `target` by reversing subarrays.

Following the solution approach:

1. **Sorting the arrays:** We start by sorting both `target` and `arr`.

   After sorting, `target` remains `[1, 2, 3, 4]` because it was already sorted. The `arr` array after sorting becomes `[1, 2, 3, 4]`.

2. **Comparing the sorted arrays:** Now, we compare the sorted `target` array with the sorted `arr` array.

   Since `sorted_target == [1, 2, 3, 4]` and `sorted_arr == [1, 2, 3, 4]`, comparison shows that both arrays are identical.

3. **Result:** As the sorted arrays are identical, we can conclude that it is possible to make `arr` identical to `target` by reversing subarrays.

Hence, according to the solution approach outlined in the problem content, the function would return `true` for these arrays.

Python Solution

```python
from typing import List  # Import List from typing module for type annotations

class Solution:
    def canBeEqual(self, target: List[int], arr: List[int]) -> bool:
        """
        Check if two lists, target and arr, can be made equal through sorting.

        Args:
        target (List[int]): The target list that arr should match.
        arr (List[int]): The list to compare with the target list.

        Returns:
        bool: True if arr can be made equal to target by sorting, False otherwise.
        """
        # Sort both the target list and arr list in place
        target.sort()
        arr.sort()

        # After sorting, if target is equal to arr, it means arr can be made equal
        # to target by sorting. Otherwise, it's not possible.
        return target == arr

```

Java Solution

```java
class Solution {

    public boolean canBeEqual(int[] target, int[] arr) {
        // Sort the target array in-place
        Arrays.sort(target);
        // Sort the arr array in-place
        Arrays.sort(arr);

        // Check if the sorted arrays are equal
        // The equals method checks if the two arrays have the same elements in the same order
        return Arrays.equals(target, arr);
    }
}

```

C++ Solution

```cpp
#include <vector> // Include vector header for using vectors
#include <algorithm> // Include algorithm header for using sort function

class Solution {
public:
    // Method to determine if two vectors can be made equal by reordering
    bool canBeEqual(vector<int>& target, vector<int>& arr) {
        // Sort the target vector in non-decreasing order
        sort(target.begin(), target.end());

        // Sort the arr vector in non-decreasing order
        sort(arr.begin(), arr.end());

        // Compare the sorted vectors to check if they are equal
        return target == arr;
    }
};

```

Typescript Solution

```typescript
function canBeEqual(target: number[], arr: number[]): boolean {
    // Determine the length of the 'target' array.
    const arrayLength = target.length;

    // Initialize an array for counting occurrences with fixed size 1001, filled with zeros.
    // This is based on the constraint that the elements in 'target' and 'arr' are integers between 1 and 1000.
    const occurrenceCount = new Array(1001).fill(0);

    // Iterate over each element of 'target' and 'arr'.
    for (let index = 0; index < arrayLength; index++) {
        // Increment the count for the current element in 'target'.
        occurrenceCount[target[index]]++;
        // Decrement the count for the current element in 'arr'.
        occurrenceCount[arr[index]]--;
    }

    // Check if every value in our counting array is zero.
    // If so, this means 'target' and 'arr' have the same elements with the same quantity.
    return occurrenceCount.every(value => value === 0);
}

```

Time and Space Complexity

Time Complexity

The given code consists of two sort operations and one equality check operation. The sort operations on both `target` and `arr` are the dominant factors in the time complexity of this function.

Assuming that the sort function is based on an algorithm with O(n log n) time complexity such as Timsort (which is the sorting algorithm used by Python's built-in sort method), the time complexity for sorting both lists would be `O(n log n)`, where `n` is the length of the lists.

The time complexity would be the sum of sorting the two lists:

- First sort: `O(target.length log(target.length))`
- Second sort: `O(arr.length log(arr.length))`

Since the question implies that both lists should be of the same length for them to be possibly equal, we can assume `target.length == arr.length` and use `n` as the length for both:

- Total time for sorting: `2 * O(n log n)`

The equality check operation that follows (`target == arr`) compares each element between the two lists, which has a time complexity of `O(n)`.

However, since the time for sorting (`O(n log n)`) is greater than the time for comparison (`O(n)`), the overall time complexity of the function is dominated by the sorting time:

- Overall time complexity: `O(n log n)`

Space Complexity

Considering the space complexity, the sort operations are done in-place in Python, which means that no additional space proportional to the input size is required beyond a constant amount used by the sorting algorithm itself.

Thus, the space complexity of the function is:

- Space complexity: `O(1)` (constant space complexity, not counting the input and output)