1122. Relative Sort Array

Problem Description

In this problem, we are provided with two integer arrays, `arr1` and `arr2`. The elements within `arr2` are distinct, which means no element repeats itself, and all these elements are present within `arr1`. Our goal is to sort `arr1` so that the order of elements matches how they appear in `arr2`, while any elements that are not found in `arr2` should be appended to the end of `arr1` in ascending order.

For example, if we have `arr1 = [2,3,1,3,2,4,6,7,9,2,19]` and `arr2 = [2,1,4,3,9,6]`, the first step is to arrange the elements of `arr1` that are also in `arr2` in the order they appear in `arr2`: `[2,2,2,1,4,3,3,9,6]`. After that, we take the remaining elements that are not in `arr2` (which are `7,19`, and `7` in this case) and place them at the end in sorted order: `[7,19,7]`. The final output array would be the concatenation of these two processes, giving us `[2,2,2,1,4,3,3,9,6,7,7,19]`.

The challenge lies in figuring out a strategy that allows us to sort `arr1` with these constraints in mind.

Intuition

The key to solving this problem efficiently is to recognize that we can use a custom sorting strategy. The default sort mechanisms most programming languages provide can often be customized with a user-defined key function that determines the sort order.

Given that the elements in `arr2` are the first to be sorted and they are in a specific order, we need to have this order play a significant role in our custom sorting strategy. For elements that are in `arr2`, we want them to appear first and in the order they are found in `arr2`.

To achieve this, we can map each element of `arr1` to its position (or index) within `arr2`. This can be done using a dictionary or a hash map that takes the element as a key and the position of that element in `arr2` as the value.

In Python, this mapping can be created as follows: `pos = {x: i for i, x in enumerate(arr2)}`

For the elements of `arr1` that are not found in `arr2`, we need to ensure they come after any element that is in `arr2`. One way to do this is to assign them an index that is larger than any index that would be assigned to the elements that are in `arr2`. Since indices in Python are zero-based and `arr2` can have, at most, the same number of elements as `arr1`, all the elements not in `arr2` can be given an index based on a large number (for example, 1000) plus their value, which ensures they will be sorted in ascending order after the elements of `arr2`.

Now, using Python's `sorted` function, we can sort `arr1` with a custom key. This key is defined by a lambda function that takes an element `x` and returns its corresponding index from the dictionary `pos` if it's in `arr2`, or `1000 + x` otherwise.

Here's the implementation in Python, which reflects our approach:

```
1  sorted(arr1, key=lambda x: pos.get(x, 1000 + x))
```

By using this custom sorting key, we align `arr1` elements per `arr2`'s order and then append the remaining elements in ascending order right after. The overall runtime complexity of the solution is O(NlogN), where N is the length of `arr1`.

Solution Approach

The solution is implemented using a dictionary for direct mapping and the sorting algorithm that Python provides. Here's a step-by-step explanation, referencing the provided code:

1. **Create a Mapping for `arr2` Elements:** The solution starts by creating a dictionary called `pos`. This dictionary maps each element `x` of `arr2` to its index `i` in the array. This is done using a dictionary comprehension:

   ```
   1  pos = {x: i for i, x in enumerate(arr2)}
   ```

   By doing this, we are setting up a quick reference so that we can look up the index of any element from `arr2` instantly. This index acts as a custom priority value for the sorting algorithm.

2. **Custom Sort with a Lambda Function:** We need to sort `arr1`, but not by its elements' natural order. We will use Python's `sorted` function, which allows us to specify a custom key function:

   ```
   1  sorted(arr1, key=lambda x: pos.get(x, 1000 + x))
   ```

   The `key` parameter is crucial here. It takes a lambda function which determines how the elements of `arr1` should be compared during sorting:

   - `lambda x:` is an anonymous function that takes `x`, an element from `arr1`.
   - `pos.get(x, 1000 + x)` is the function's body that returns a value for each `x` that will be used to compare during sorting.
   - If `x` is found in `arr2` (and, as a result, the `pos` dictionary), `pos.get(x)` returns the index of `x` from `arr2`.
   - If `x` is not found in `arr2`, the method `pos.get(x, 1000 + x)` provides a default value which is the sum of 1000 and the element's value (`x`). This ensures that:
     - The elements not in `arr2` are ordered at the end because their sorting key is greater than any index derived from `arr2`.
     - They are sorted in ascending order amongst themselves because if two elements are not found in `arr2`, their sorting key is simply their value (`1000 + their own value` respects the natural ascending order).

By using this sorting strategy, we can maintain the relative ordering based on another array `arr2` while also logically appending and sorting elements not found in `arr2`.

Thus, using a combination of dictionary for direct access and sorting algorithm with a custom key function, this solution efficiently achieves the required array manipulation adhering to the conditions of the problem statement.

Example Walkthrough

Let's illustrate the solution approach using a smaller example. Suppose `arr1` contains `[8, 4, 5, 4, 6]` and `arr2` contains `[4, 6]`. We want to sort `arr1` such that the order of `arr2` is preserved for common elements, and the rest are sorted in ascending order at the end.

Step 1: Create a Mapping for `arr2` Elements:

First, we create a dictionary that maps each element of `arr2` to its index:

```
1  pos = {x: i for i, x in enumerate(arr2)}
2  # pos will be {4: 0, 6: 1}
```

Here, `4` is at index `0` in `arr2` and `6` is at index `1`.

Step 2: Sort `arr1` with a Custom Sort Key:

We now sort `arr1` using Python's `sorted` function with the custom key:

```
1  sorted(arr1, key=lambda x: pos.get(x, 1000 + x))
```

Breaking down how the key function works for each element in `arr1`:

- `8` is not found in `arr2`, so the key would be `1000 + 8 = 1008`.
- The first `4` is found in `arr2` and its index is `0`; hence, the key would be `0`.
- `5` is not found in `arr2`, so its key is `1000 + 5 = 1005`.
- The second `4` is found and its key is again `0`.
- `6` is found and its index is `1`, so the key is `1`.

The sorted order using these keys would be `[4, 4, 6, 5, 8]`.

Step 3: Review the Final Sorted Array:

So, the elements from `arr1` that were also in `arr2` are now sorted in `arr2` order: `4` comes first, followed by `6`. The remaining elements, not in `arr2`, follow at the end in ascending order: `5` and then `8`.

In conclusion, after applying the custom key sorting strategy to `arr1`, we get the final sorted array: `[4, 4, 6, 5, 8]`, successfully matching the required conditions.

Python Solution

```
1  class Solution:
2      def relative_sort_array(self, arr1: List[int], arr2: List[int]) -> List[int]:
3          # Create a dictionary to map elements of arr2 to their indices.
4          position_map = {value: index for index, value in enumerate(arr2)}
5
6          # Sort arr1 based on the condition that elements of arr2 should come first
7          # in the order they appear in arr2, followed by the remaining elements in
8          # ascending order.
9          sorted_arr1 = sorted(arr1, key=lambda x: position_map.get(x, 1000 + x))
10
11         # The lambda function inside sorted uses the 'get' method to find the position
12         # of x from position_map if x exists in arr2. Otherwise, it assigns a number
13         # greater than 1000 to ensure that x is positioned after all elements of arr2.
14         # This is based on the assumption that the elements of arr1 do not exceed 1000.
15
16         return sorted_arr1
17
```

Java Solution

```
1  class Solution {
2      public int[] relativeSortArray(int[] arr1, int[] arr2) {
3          // Create a hashmap to store the positions of each element in arr2
4          Map<Integer, Integer> elementToIndexMap = new HashMap<>(arr2.length);
5          // Fill the map with the positions of the elements
6          for (int index = 0; index < arr2.length; ++index) {
7              elementToIndexMap.put(arr2[index], index);
8          }
9
10         // Create a new 2D array to hold elements and their corresponding positions
11         int[][] elementPositionPairs = new int[arr1.length][2];
12         for (int index = 0; index < elementPositionPairs.length; ++index) {
13             // Use the position from arr2 if present, or else use the value from arr1 plus the length of arr2
14             elementPositionPairs[index] = new int[]{arr1[index], elementToIndexMap.getOrDefault(arr1[index], arr2.length + arr1[index
15         }
16
17         // Sort the 2D array based on the positions
18         Arrays.sort(elementPositionPairs, (pair1, pair2) -> pair1[1] - pair2[1]);
19
20         // Place the sorted elements back into arr1
21         for (int index = 0; index < elementPositionPairs.length; ++index) {
22             arr1[index] = elementPositionPairs[index][0];
23         }
24
25         // Return the sorted arr1
26         return arr1;
27     }
28 }
```

C++ Solution

```
1  #include <vector>
2  #include <unordered_map>
3  #include <algorithm>
4
5  class Solution {
6  public:
7      // Function to sort arr1 with respect to the order in arr2
8      std::vector<int> relativeSortArray(std::vector<int>& arr1, std::vector<int>& arr2) {
9          // We will use an unordered_map to store the positions (index) of
10         // each element in arr2; this position is used when sorting arr1
11         std::unordered_map<int, int> elementToIndex;
12         for (int i = 0; i < arr2.size(); ++i) {
13             elementToIndex[arr2[i]] = i; // Map the element to its index in arr2
14         }
15
16         // Vector to temporarily store pairs of position and value
17         // from original array arr1. If the element does not exist in arr2,
18         // we assign the index as the size of arr2 which is effectively
19         // putting it at the end of the sorted array.
20         std::vector<std::pair<int, int>> tempArray;
21         for (int value : arr1) {
22             // The index (if value exists in arr2; for elements not in arr2, set index to arr2's size
23             int index = elementToIndex.count(value) ? elementToIndex[value] : arr2.size();
24             // Create a pair of (index, value) for each element in arr1
25             tempArray.emplace_back(index, value);
26         }
27
28         // Sort the tempArray based on the previously stored positions (indices)
29         // If two elements have the same position, they're compared by their values
30         std::sort(tempArray.begin(), tempArray.end());
31
32         // Reassign sorted values back to arr1
33         for (int i = 0; i < arr1.size(); ++i) {
34             arr1[i] = tempArray[i].second; // Set arr1[i] with the value from the sorted pair
35         }
36
37         // Return the sorted arr1
38         return arr1;
39     }
40 };
```

Typescript Solution

```
1  // Given two arrays arr1 and arr2, the function sorts elements of arr1
2  // in such that the relative ordering of items in arr1 are the same as in arr2.
3  // Elements not present in arr2 will be placed at the end of arr1 in ascending order.
4  function relativeSortArray(arr1: number[], arr2: number[]): number[] {
5      // Create a mapping of element-to-index position for quick access
6      const elementToIndex: Map<number, number> = new Map();
7      // Fill the map with elements of arr2 and their corresponding indices
8      for (let index = 0; index < arr2.length; ++index) {
9          elementToIndex.set(arr2[index], index);
10     }
11
12     // Initialize an array to hold pairs of position in arr2 and the value
13     const sortedPairs: [number, number][] = [];
14
15     // Map each element of arr1 to its corresponding index in arr2, with a fallback index
16     for (const element of arr1) {
17         // Get the index position of the element from the map or use default if not present
18         const indexInArr2 = elementToIndex.get(element) ?? arr2.length;
19         sortedPairs.push([indexInArr2, element]);
20     }
21
22     // Sort the array: first by the index position and then by the element values
23     sortedPairs.sort((a, b) => a[0] - b[0] || a[1] - b[1]);
24
25     // Map the sorted array back to a single array of values
26     return sortedPairs.map(pair => pair[1]);
27 }
```

Time and Space Complexity

Time Complexity

The time complexity of the given code primarily depends on the sorting algorithm used by the `sorted` function in Python.

- Creating the `pos` dictionary has a time complexity of O(M) where M is the length of `arr2`, since we iterate over each element of `arr2` once.
- The `sorted` function has a time complexity of O(N log N) where N is the length of `arr1`. This is because Python uses the TimSort algorithm for sorting, which is a hybrid sorting algorithm derived from merge sort and insertion sort.

However, since the key function in the sorted algorithm uses a lookup in the `pos` dictionary, which is O(1) time complexity for each element, together with the condition to handle elements not in `pos`, the overall time complexity of the sorting with these operations is still O(N log N).

Therefore, the total complexity of the algorithm is O(M + N log N).

Space Complexity

The space complexity of the algorithm consists of the space needed for the `pos` dictionary and any additional space used by the sorting algorithm.

- The `pos` dictionary has a space complexity of O(M) because it contains as many entries as there are elements in `arr2`.
- The `sorted` function returns a new list and internally uses additional space which depends on the specifics of the implementation. In the worst case, this could be O(N) space.

The total space complexity is therefore O(M + N).