2089. Find Target Indices After Sorting Array

Easy Array Binary Search Sorting

Problem Description

In this problem, we are given an array of integers called `nums` and another integer called `target`. Our task is to find all the indices in the array where the element is equal to the target, after sorting the array in non-decreasing order (from smallest to largest values). The "target indices" are the positions in the sorted array where the target is found. We're required to return these indices as a list, which should also be sorted in increasing order. If the target is not present in the array, we should return an empty list.

Intuition

To solve this problem, the intuitive approach is straightforward:

1. Sort the array in non-decreasing order so that any duplicates of `target` will be positioned next to each other.
2. Iterate through the sorted array and for each element that is equal to `target`, record its index.

By doing these steps, we ensure that we're considering the elements in the sorted order and collecting the indices of the target. Since we are sorting the array first, the indices that we collect will already be sorted. Thus, the resulting list of indices satisfies the problem's requirements without needing further sorting.

The Python solution provided leverages list comprehension, which is a concise way to iterate over the sorted array and construct the list of target indices in one go.

Solution Approach

The implementation of the solution follows a simple algorithm that involves sorting and then iterating through the array. The two main components of the implementation are the sorting algorithm and the enumeration pattern, which are both native to Python.

Here's the breakdown of the solution approach:

1. `nums.sort()`: The `sort()` method is called on the `nums` array. In Python, this method uses a TimSort algorithm, which is a hybrid sorting algorithm derived from merge sort and insertion sort. It has a time complexity of `O(n log n)` on average, where `n` is the number of elements in the array. This step rearranges the elements of `nums` in-place in a non-decreasing order.
2. `[i for i, v in enumerate(nums) if v == target]`: This is a list comprehension that creates a new list. The `enumerate(nums)` function is used to get both the index (`i`) and the value (`v`) of each element in the sorted `nums` array. The `if v == target` part is a condition that filters out all the elements that are not equal to the `target`. Only the indices of the elements that match the `target` are included in the final list.

The algorithm's space complexity is `O(1)` for the sorting (since it sorts in-place), and the space complexity for the list comprehension is `O(k)`, where `k` is the number of times the target appears in `nums` since it creates a new list that contains all the target indices.

Overall, the solution is efficient and leverages Python's built-in functions to achieve the desired result with concise and readable code.

Example Walkthrough

Let's go through an example to illustrate the solution approach.

Suppose we have an array of integers `nums = [4, 1, 2, 1, 3, 2]` and the target integer `target = 2`. We want to find all indices of `target` in the sorted array.

Here is how we apply the solution approach step-by-step:

1. Sort the Array:

   We first sort the array in non-decreasing order. Applying `nums.sort()` will modify our `nums` array to `[1, 1, 2, 2, 3, 4]`.

2. Find Target Indices:

   We then use list comprehension to find all indices where the value is equal to `target`. For our sorted array, it would look like this:

   ```
   [i for i, v in enumerate([1, 1, 2, 2, 3, 4]) if v == 2]
   ```

   Here, `enumerate()` function gives us pairs of indices and their corresponding values. We only want the indices where the value `v` is equal to our target, which is `2`.

3. Filtering:

   As we iterate, we check each value `v`:

   - At index 0, `v` is `1`, which is not equal to `2`.
   - At index 1, `v` is `1`, which is also not equal to `2`.
   - At index 2, `v` is `2`, which is equal to `2`. We add `2` to our list.
   - At index 3, `v` is `2`, which is again equal to `2`. We add `3` to our list.
   - The last two values at indices 4 and 5 are `3` and `4`, neither of which matches our target.

4. Final Result:

   The resulting list of target indices after applying the filter is `[2, 3]`, which are the sorted indices in the original sorted `nums` array where the target value `2` is located.

Thus, if we call our function with the above `nums` and `target`, we will get `[2, 3]` as the output because these are the positions in the sorted array where `2` is found.

Solution Implementation

Python

```python
class Solution:
    def target_indices(self, numbers: List[int], target: int) -> List[int]:
        # Sort the list of numbers in place
        numbers.sort()

        # Use list comprehension to find all indices where the value equals the target
        # This loop iterates over each index and value in the sorted list of numbers
        target_indices_list = [index for index, value in enumerate(numbers) if value == target]

        return target_indices_list
```

Java

```java
class Solution {
    public List<Integer> targetIndices(int[] nums, int target) {
        // Sort the array in non-decreasing order
        Arrays.sort(nums);

        // Initialize an empty list to hold the indices of the target
        List<Integer> targetIndicesList = new ArrayList<>();

        // Loop through the sorted array
        for (int index = 0; index < nums.length; index++) {
            // If the current element is equal to the target...
            if (nums[index] == target) {
                // ...then add its index to the list
                targetIndicesList.add(index);
            }
        }

        // Return the list of indices where the target is found
        return targetIndicesList;
    }
}
```

C++

```cpp
#include <vector>
#include <algorithm>

class Solution {
public:
    // Function to find all indices of 'target' in a sorted vector 'nums'.
    std::vector<int> targetIndices(std::vector<int>& nums, int target) {
        // First, sort the given vector.
        std::sort(nums.begin(), nums.end());

        // Declare a vector to store the indices where 'target' is found.
        std::vector<int> result_indices;

        // Iterate through the sorted vector to find all occurrences of 'target'.
        for (int index = 0; index < nums.size(); ++index) {
            // If the current element equals 'target', add its index to the result.
            if (nums[index] == target) {
                result_indices.push_back(index);
            }
        }

        // Return the vector containing all indices of 'target'.
        return result_indices;
    }
};
```

TypeScript

```typescript
// Function to find all indices at which a given target number appears
// after sorting the array in ascending order.
function targetIndices(nums: number[], target: number): number[] {
    // Sort the array in ascending order.
    nums.sort((a, b) => a - b);

    // Initialize an array to store the indices where target is found.
    let resultIndices: number[] = [];

    // Iterate over the sorted array to find all occurrences of target.
    for (let index = 0; index < nums.length; index++) {
        // Check if the current element is equal to the target.
        if (nums[index] === target) {
            // If it is, add the current index to the resultIndices array.
            resultIndices.push(index);
        }
    }

    // Return the array of indices where target is found.
    return resultIndices;
}
```

```python
class Solution:
    def target_indices(self, numbers: List[int], target: int) -> List[int]:
        # Sort the list of numbers in place
        numbers.sort()

        # Use list comprehension to find all indices where the value equals the target
        # This loop iterates over each index and value in the sorted list of numbers
        target_indices_list = [index for index, value in enumerate(numbers) if value == target]

        return target_indices_list
```

Time and Space Complexity

Time Complexity

The time complexity of the provided code primarily comes from the `sort` method which has a time complexity of `O(n log n)` where `n` is the length of the `nums` list. After sorting, the code iterates through the list to find indices of elements equal to the target with a time complexity of `O(n)`. Therefore, the total time complexity is `O(n log n) + O(n)`, which simplifies to `O(n log n)` since `n log n` dominates for larger values of `n`.

Space Complexity

The space complexity of the code is `O(1)` if we use the sorting algorithm that sorts the input list in place, such as Timsort (which Python's `sort` method uses). No additional space is required proportional to the input size other than the space for the output list. The list comprehension for the indices generates the output list, so its space is necessary for the result and doesn't count towards auxiliary space complexity.