1636. Sort Array by Increasing Frequency

Easy | Array | Hash Table | Sorting

Problem Description

The problem provides us with an array of integers `nums`. The task is to sort the array, but not by the usual ascending or descending orders based purely on the number values. Instead, we need to first sort the array based on how frequent each number appears, which we'll call its frequency. So, numbers with a lower frequency should come first. In the event that two numbers have the same frequency, we need to sort these numbers in *decreasing* order amongst themselves. Finally, we'll return the rearranged array that satisfies these sorting rules.

Intuition

To solve this problem, one strategy is to lean on Python's sorting capabilities but with a custom sorting rule. We need to be able to sort by two criteria: frequency, and then by the value itself if frequencies are equal. In Python, we can achieve this by using a sorting function that allows a custom key, to which we can pass a lambda function.

The lambda function will return a tuple with two elements: the frequency of the number and the negation of the number itself. The reason for negating the number is because Python sorts tuples in lexicographical order, starting with the first element. For the first element, we want a normal increasing order based on frequency, but for the second element, we want decreasing order. Since Python will normally sort in increasing order, negating the numbers allows us to "invert" this to get the desired decreasing order. The standard `sorted` function in Python will then take care of the rest, applying our custom key to sort the array as required by the problem.

We use the `Counter` class from the collections module to quickly tally up the frequencies of each element in the `nums` array. This construct, when used in the `sorted` function's key, performs the sort based on the aforementioned tuple, `(frequency, -value)`.

Solution Approach

The solution uses a combination of a custom sorting function and a hash table, provided by Python's `Counter` from the collections module, to count the frequency of each element in the `nums` array.

Here's a step-by-step walkthrough:

1. The `Counter(nums)` function creates a hash table that maps each unique number in the `nums` array to its frequency. Let's call this map `cnt`.
2. The built-in `sorted` function in Python is used to sort the numbers, but instead of sorting by the numbers themselves, it sorts by a key that we define. This key is a lambda function, which Python allows for customization of the sort order.
3. The lambda function takes an element `x` from `nums` and returns a tuple: `(cnt[x], -x)`. Here, `cnt[x]` is the frequency of `x`, so the first element of the tuple dictates that the sorter first arranges elements based on increasing frequency (lower frequency comes first).
4. The second element of the tuple is `-x`, the negation of the number itself. This part ensures that if two elements have the same frequency (`cnt[x]` is equal), the sorter will arrange these particular items based on their value in descending order (since `-x` converts the sort order).

To clarify via an example, let's say `nums` contains `[1, 1, 2, 2, 3, 3, 3]`. The `Counter` would yield `cnt` as `{1: 2, 2: 2, 3: 3}`. The sorted key would transform these into sort keys `[2, -1]`, `[2, -1]`, `[2, -2]`, `[2, -2]`, `[3, -3]`, `[3, -3]`, `[3, -3]`. The sorted function would use these keys to sort `nums` into `[3, 3, 3, 2, 2, 1, 1]`.

In the end, the lambda function in conjunction with `sorted()` applies a sorting algorithm that respects the two key sorting rules outlined in the problem description, effectively yielding the desired sorted array.

Example Walkthrough

Let's consider a small example with the array `nums = [4, 1, 6, 6, 4, 4, 6]` to illustrate the solution approach.

1. First, we apply the `Counter(nums)` function from the collections module to count the frequency of each unique number in `nums`. This gives us a frequency map `cnt` like so: `{4: 3, 1: 1, 6: 3}`.
2. We then call the built-in `sorted` function with a custom key. The key is defined by a lambda function such that each element `x` from `nums` is transformed into a sort key `(cnt[x], -x)`.
3. Applying this lambda function to each element of `nums` gives us the following sort keys: `(3, -4)` for each 4, `(1, -1)` for the 1, and `(3, -6)` for each 6. The sort keys are derived from the `cnt` map and negation of the value as per our lambda function.
4. These sort keys are then used by the `sorted` function to sort `nums`. The first element of the tuple determines that we sort by increasing frequency, with lower frequencies coming first. The array element with the lowest frequency (1 in this case) will thus come first in the sorted array.
5. For elements with the same frequency (4 and 6 in this case), the second element of the tuple comes into play, sorting these in decreasing order due to the negation (`-x`). This means that between 4 and 6, which have the same frequency, 6 will come before 4 in the sorted array.

In practice, the `sorted` function sorts the keys as follows: `[ (1, -1), (3, -6), (3, -6), (3, -6), (3, -4), (3, -4), (3, -4) ]`. Once translated back into the actual `nums` values, we get the final sorted array: `[1, 6, 6, 6, 4, 4, 4]`.

We have successfully walked through the steps of the approach using a small example, which shows how the array is transformed according to the given two-step sorting process: first by increasing frequency of the elements, then by their values in decreasing order where frequencies are equal.

Solution Implementation

Python

```python
from collections import Counter

class Solution:
    def frequencySort(self, nums: List[int]) -> List[int]:
        # Count the frequency of each number in the list using Counter
        num_frequency = Counter(nums)

        # Sort the numbers based on the frequency (ascending order)
        # When frequencies are the same, sort by the number itself (descending order)
        sorted_nums = sorted(nums, key=lambda x: (num_frequency[x], -x))

        return sorted_nums
```

Java

```java
class Solution {
    public int[] frequencySort(int[] nums) {
        // Array to keep track of the frequency of each number.
        // Since the range of numbers is from -100 to 100, an offset of 100 is used
        // to map them to indices 0 to 200.
        int[] frequency = new int[201];
        // Transform the input array into a list to facilitate custom sorting.
        List<Integer> transformedList = new ArrayList<>();

        // Count frequencies and populate the list.
        for (int num : nums) {
            num += 100; // Apply offset to handle negative values.
            ++frequency[num]; // Increment frequency count for this number.
            transformedList.add(num); // Add to transformed list.
        }

        // Sort the list first by frequency, then by value if frequencies are equal.
        transformedList.sort((a, b) ->
            // If frequencies are the same, sort in descending order of the values (b - a).
            // Otherwise, sort by ascending order of frequencies (cnt[a] - cnt[b]).
            frequency[a] == frequency[b] ? b - a : frequency[a] - frequency[b]
        );

        // Create an array to store the sorted values.
        int[] sortedArr = new int[nums.length];
        int i = 0;
        // Populate the sortedArr with sorted values from the list,
        // converting them back by removing the offset of 100.
        for (int val : transformedList) {
            sortedArr[i++] = val - 100;
        }
        return sortedArr; // Return the sorted array.
    }
}
```

C++

```cpp
#include <vector>   // Include the necessary header for using the vector container
#include <algorithm> // Include the algorithm header for the sort function

class Solution {
public:
    vector<int> frequencySort(vector<int>& nums) {
        // Vector to store the frequency of numbers
        // 201 in size to account for numbers from -100 to 100 (inclusive)
        vector<int> frequency(201, 0);

        // Counting each number's frequency
        // Shift the index to fit in the range of [0, 200]
        for (int num : nums) {
            ++frequency[num + 100];
        }

        // Custom sort the numbers
        sort(nums.begin(), nums.end(), [&](const int num1, const int num2) {
            // When frequencies are equal, sort by number value in descending order
            if (frequency[num1 + 100] == frequency[num2 + 100]) return num1 > num2;
            // Otherwise, sort by frequency in ascending order
            return frequency[num1 + 100] < frequency[num2 + 100];
        });

        // Return the sorted vector
        return nums;
    }
};
```

TypeScript

```typescript
// Defines a function to sort an array of numbers based on frequency of each number,
// and for numbers with the same frequency, sort them in descending order.
function frequencySort(nums: number[]): number[] {
    // Create a Map to keep track of the frequency of each number.
    const frequencyMap = new Map<number, number>();
    // Iterate through the array of numbers.
    for (const num of nums) {
        // Update the Map with the new frequency of the current number.
        frequencyMap.set(num, (frequencyMap.get(num) ?? 0) + 1);
    }
    // Sort the numbers array.
    return nums.sort((a, b) => {
        // Retrieve the frequency of both numbers being compared.
        const freqA = frequencyMap.get(a) ?? 0;
        const freqB = frequencyMap.get(b) ?? 0;
        // If frequencies are different, sort by frequency in ascending order.
        if (freqA !== freqB) {
            return freqA - freqB;
        }
        // If frequencies are the same, sort by number in descending order.
        return b - a;
    });
}
```


```python
from collections import Counter

class Solution:
    def frequencySort(self, nums: List[int]) -> List[int]:
        # Count the frequency of each number in the list using Counter
        num_frequency = Counter(nums)

        # Sort the numbers based on the frequency (ascending order)
        # When frequencies are the same, sort by the number itself (descending order)
        sorted_nums = sorted(nums, key=lambda x: (num_frequency[x], -x))

        return sorted_nums
```


Time and Space Complexity

The given Python code sorts an array of integers based on the frequency of each number and uses the `Counter` from the `collections` module to count occurrences. Then, it employs a custom sorting function.

Time Complexity:

The time complexity of the code is determined by several operations:

1. Counting the occurrences with `Counter`: This has a time complexity of `O(n)` where `n` is the number of elements in `nums`.
2. Sorting with the sorted function: Python's sorting function uses the Timsort algorithm, which has a time complexity of `O(n log n)` in the average and worst case.
3. Custom sorting function: Sorting based on two criteria (count and then number itself in the reverse order) does not change the overall time complexity of `O(n log n)`.

Therefore, considering these together, the total time complexity is `O(n + n log n)`, which simplifies to `O(n log n)` since `n log n` is the dominating term.

Space Complexity:

The space complexity is determined by:

1. Counter dictionary: The `Counter` creates a dictionary with as many entries as the unique elements in `nums`, which in the worst case is `O(n)`.
2. Sorted function: Timsort requires `O(n)` space in the worst case.

Thus, the overall space complexity of the function is `O(n)` where `n` is the number of elements in `nums`.