<https://leetcode.com/problems/sort-array-by-increasing-frequency/description/>

Given an array of integers nums, sort the array in **increasing** order based on the frequency of the values. If multiple values have the same frequency, sort them in **decreasing** order.

Return the **sorted array**.

**Example 1:**

**Input:** nums = [1,1,2,2,2,3]

**Output:** [3,1,1,2,2,2]

**Explanation:** '3' has a frequency of 1, '1' has a frequency of 2, and '2' has a frequency of 3.

**Example 2:**

**Input:** nums = [2,3,1,3,2]

**Output:** [1,3,3,2,2]

**Explanation:** '2' and '3' both have a frequency of 2, so they are sorted in decreasing order.

**Example 3:**

**Input:** nums = [-1,1,-6,4,5,-6,1,4,1]

**Output:** [5,-1,4,4,-6,-6,1,1,1]

**Constraints:**

1 <= nums.length <= 100

-100 <= nums[i] <= 100

**Attempt 1: 2024-01-26**

**Solution 1: Hash Table (10 min)**

class Solution {

    public int[] frequencySort(int[] nums) {

        Map<Integer, Integer> map = new HashMap<>();

        for(int num : nums) {

            map.put(num, map.getOrDefault(num, 0) + 1);

        }

        List<Map.Entry<Integer, Integer>> list = new ArrayList<>();

        for(Map.Entry<Integer, Integer> e : map.entrySet()) {

            list.add(e);

        }

        Collections.sort(list, (a, b) -> a.getValue() == b.getValue() ? b.getKey() - a.getKey() : a.getValue() - b.getValue());

        int[] result = new int[nums.length];

        int i = 0;

        for(Map.Entry<Integer, Integer> e : list) {

            int freq = e.getValue();

            int val = e.getKey();

            while(freq > 0) {

                result[i] = val;

                freq--;

                i++;

            }

        }

        return result;

    }

}

Time Complexity: O(N\*logN)

Space Complexity: O(N)

**Refer to**

<https://leetcode.com/problems/sort-array-by-increasing-frequency/solutions/917993/java-simple-custom-sort-with-detailed-explanation/comments/929415>

/\*\*\*\*\*\*

-- Scan array and put it in a Frequency Map.

-- Construct List & Sort based on Frequency and decreasing order

-- Read the sorted List and put the elements in output array

\*\*\*\*\*/

class Solution {

public int[] frequencySort(int[] nums) {

//Scan Array and update Frequency Map

Map<Integer, Integer> map = new HashMap<Integer, Integer>();

for(int num : nums){

map.put(num, map.getOrDefault(num, 0)+1);

}

//Create List with Map.Entry

List<Map.Entry<Integer,Integer>> list = new ArrayList<>(map.entrySet());

//Sort the List based on Comparator

Collections.sort(list, new Comp());

//Create output array

int[] ans = new int[nums.length];

int index = 0;

//Populate the Output array

for(Map.Entry<Integer,Integer> entry : list){

int val = entry.getValue();

while(val > 0){

ans[index++] = entry.getKey();

val--;

}

}

return ans;

}

//Compare as per Frequency and Key.

private static class Comp implements Comparator<Map.Entry<Integer,Integer>>{

@Override

public int compare(Map.Entry<Integer,Integer> m1, Map.Entry<Integer,Integer> m2){

if(m1.getValue() == m2.getValue()){

return -m1.getKey() + m2.getKey();

}

return m1.getValue() - m2.getValue();

}

}

}

**How to sort Map ?**

**Refer to**

<https://javarevisited.blogspot.com/2017/07/how-to-sort-map-by-keys-in-java-8.html#axzz8Pys5lDiH>

In the last article, I have shown you how to sort a Map by values in Java 8, and in this tutorial, you will learn how to sort a Map by keys like a HashMap, ConcurrentHashMap, LinkedHashmap, or even Hashtable. **Theoretically, you cannot sort a Map because it doesn't provide any ordering guarantee.** For example, when you iterate over a HashMap, you don't know in which order entries will be traversed because HashMap doesn't provide any ordering. Then, how can you sort a Map which doesn't support order? **Well, you can't and that's why you only sort entries of HashMap but you don't store the result back into HasMap or any other Map which doesn't support ordering. If you do so, then sorting will be lost.**

**Solution 2: Use Arrays.stream() (10 min)**

class Solution {

    public int[] frequencySort(int[] nums) {

        Map<Integer, Integer> map = new HashMap<>();

        // count frequency of each number

        Arrays.stream(nums).forEach(n -> map.put(n, map.getOrDefault(n, 0) + 1));

        // custom sort

        return Arrays.stream(nums).boxed()

                .sorted((a, b) -> map.get(a) != map.get(b) ? map.get(a) - map.get(b) : b - a)

                .mapToInt(n -> n)

                .toArray();

    }

}

Time Complexity: O(N\*logN)

Space Complexity: O(N)

**Refer to**

<https://leetcode.com/problems/sort-array-by-increasing-frequency/solutions/917993/java-simple-custom-sort-with-detailed-explanation/>

public int[] frequencySort(int[] nums) {

    Map<Integer, Integer> map = new HashMap<>();

    // count frequency of each number

    Arrays.stream(nums).forEach(n -> map.put(n, map.getOrDefault(n, 0) + 1));

    // custom sort

    return Arrays.stream(nums).boxed()

            .sorted((a,b) -> map.get(a) != map.get(b) ? map.get(a) - map.get(b) : b - a)

            .mapToInt(n -> n)

            .toArray();

}

custom sort explanation:

**.stream(nums)**

**iterates through the nums array**

**.boxed()**

**converts each int to Integer object, this is because .sorted() can only operate on objects**

**.sorted((a,b) -> map.get(a) != map.get(b) ? map.get(a) - map.get(b) : b - a)**

**if frequency of two numbers are not the same, sort by ascending frequency. If frequencies are the same, sort by decending numeric value**

**.mapToInt(n -> n)**

**converts Integer to int**

**.toArray()**

**returns array**

**Solution 3: Frequency array instead of Hash Table (10 min)**

**因为题目已经给出了nums[i]的范围-100 <= nums[i] <= 100，所以直接可以用frequency array来处理**

class Solution {

    public int[] frequencySort(int[] nums) {

        int[] freq = new int[201];

        List<Integer> list = new ArrayList<>();

        for(int num : nums) {

            freq[num + 100]++;

            list.add(num);

        }

        Collections.sort(list, (a, b) -> (freq[a + 100] == freq[b + 100] ? b - a : freq[a + 100] - freq[b + 100]));

        int[] result = new int[nums.length];

        int i = 0;

        for(int num : list) {

            result[i++] = num;

        }

        return result;

    }

}

Time Complexity: O(N\*logN)

Space Complexity: O(N)

**Refer to**

<https://algo.monster/liteproblems/1636>

**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:

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.

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.

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).

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.

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}.

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).

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.

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.

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.

**Java Solution**

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.

    }

}

**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:

Counting the occurrences with Counter: This has a time complexity of O(n) where n is the number of elements in nums.

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.

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:

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).

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.