1338. Reduce Array Size to The Half

Medium | Greedy | Array | Hash Table | Sorting | Heap (Priority Queue) | Leetcode Link

Problem Description

The problem is about reducing the size of an integer array by removing some subsets of integers to make sure at least half of the elements in the original array are removed. The task is to find the minimum number of distinct integers needed for these subsets such that once their occurrences are removed, at least half of the array elements are gone.

To clarify with an example, if we have an array `[3,3,3,3,5,5,5,2,2,7]`, a possible solution would be choosing the subset `{3,5}` which contains 2 integers. Removing all the instances of `3` and `5` from the array would remove `3 + 3 = 6` elements from the 10-element array, thereby removing more than half of the array, which satisfies the condition.

Intuition

The intuition behind the solution approach involves the following steps:

1. Identify that we want to maximize the number of elements removed by picking the least number of distinct integers.
2. Understand that to remove the most numbers, we should start by removing the integers that appear most frequently.
3. Use a frequency counter to count the occurrences of each integer in the array.
4. Sort these integers by their frequencies in descending order, so we can remove the integers with higher frequencies first.
5. Start removing integers from the sorted list one by one, adding their frequencies to a cumulative sum (`m` in the code).
6. Each time an integer is added to the set (the subset we're removing), increase the count (`ans` in the code) that represents the size of our set.
7. Once we've removed enough numbers such that the cumulative sum is at least half of the original array's size, we've found our minimum set size and can stop.

The provided Python code uses a `Counter` from the `collections` module to tally the integer frequencies and `most_common()` method to sort them by frequency in descending order. It iterates through these frequencies while keeping a running total and a count until at least half the array size is reached.

Solution Approach

The solution uses a hash map and sorting technique to solve the problem efficiently. Here's a breakdown of the implementation steps referring to the Python code provided:

1. **Hash Map (Counter):**

   - Use the Python `Counter` from the `collections` module to create a hash map that counts the occurrences of each integer in the input array `arr`. This operation has a complexity of O(n), where n is the size of the array.

   ```
   cnt = Counter(arr)
   ```

2. **Sorting:**

   - The `most_common()` function is called on the `Counter` object which under the hood performs a sort operation, arranging the counted integers in descending order based on their frequency. The complexity of this operation is O(u log u), where u is the number of unique elements in `arr`.

   ```
   for _, v in cnt.most_common():
   ```

3. **Iterative Accumulation:**

   - Initialize two variables, `ans` to count the number of elements in the required set and `m` to keep track of the cumulative frequency of elements removed from the array `arr`.

   ```
   ans = m = 0
   ```

   - Iteratively add the frequency `v` of the most common element to `m`. Increment `ans` by 1 to reflect that an element is added to the set.

   ```
   m += v
   ans += 1
   ```

   - Check if the cumulative frequency `m` is at least half the length of the original array. If so, break out of the loop since the requirement of removing at least half of the array elements is met.

   ```
   if m * 2 >= len(arr):
       break
   ```

4. **Return the Result:**

   - After the loop terminates, return the value of `ans`, which represents the minimum size of the set to remove at least half of the array elements.

   ```
   return ans
   ```

This approach is efficient as it targets the frequency of elements which is a key insight for achieving the task with a minimal set size. The heavy-lifting operations are counting and sorting which are both done optimally using Python's built-in data structures and algorithms.

Example Walkthrough

Let's use a small example to illustrate the solution approach. Suppose we have an array `arr = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]`.

1. **Hash Map (Counter):**

   - First, we'll count the occurrences of each integer:

   ```
   from collections import Counter
   cnt = Counter(arr)
   # cnt = Counter({4: 4, 3: 3, 2: 2, 1: 1})
   ```

   In `cnt`, the key is the integer and the value is its frequency in the array.

2. **Sorting:**

   - Next, we sort these by frequency using `most_common()`:

   ```
   # This will sort the integers by frequency: [4, 3, 2, 1]
   sorted_integers_by_freq = [item for item, count in cnt.most_common()]
   ```

3. **Iterative Accumulation:**

   - Initialize the variables `ans` and `m`:

   ```
   ans = m = 0
   ```

   - Remove integers one by one, starting from the most frequent. In our list, `4` occurs the most, so we start with that:

   ```
   m += cnt[4]   # m = 4 (since 4 appears 4 times)
   ans += 1      # ans = 1 (we have added one integer to our set)
   ```

   - Now we move to the next most frequent integer, `3`:

   ```
   m += cnt[3]   # m = 7 (4 from '4's and 3 from '3's)
   ans += 1      # ans = 2 (we have added two integers to our set)
   ```

4. **Checking the Cumulative Frequency:**

   - We need to check if we have removed at least half of the elements in the array. The original array has 10 elements, so we need to remove at least 5. With `m = 7`, we have surpassed this:

   ```
   if m * 2 >= len(arr):  # 7*2 >= 10 which is True
       # Break the loop as we have reached our requirement.
   ```

5. **Return the Result:**

   - We've found our answer, which is 2, because we were able to remove at least half of the elements by removing numbers `4` and `3`.

   ```
   return ans
   # return 2
   ```

In this example, the subsets `{4, 3}` contain the minimum number of distinct integers (2 integers) that, once removed, reduce the original array size at least by half. Following this procedure guarantees that we achieve the goal with the smallest possible set, efficiently utilizing the insights about frequency of elements.

Python Solution

```python
from typing import List
from collections import Counter

class Solution:
    def minSetSize(self, arr: List[int]) -> int:
        # Count the frequency of each element in the array
        frequency_counter = Counter(arr)

        # Initialize variables for the minimum set size and the count of removed elements
        min_set_size = 0
        removed_count = 0

        # Iterate over the elements in the frequency counter, starting with the most common
        for _, frequency in frequency_counter.most_common():
            # Increment the count of removed elements by the frequency of the current element
            removed_count += frequency

            # Increment the set size by 1 (since we're adding one more element to the removal set)
            min_set_size += 1

            # Check if we have removed at least half of the elements from the original array
            if removed_count * 2 >= len(arr):
                # If true, we break the loop as we have reached the required set size
                break

        # Return the minimum set size needed to remove at least half of the elements
        return min_set_size

# Example usage:
# solution = Solution()
# result = solution.minSetSize([3,3,3,3,5,5,5,2,2,7])
# print(result) # Output should be 2, as removing 3 and 5 removes 6 elements, > half of the array size.

```

Java Solution

```java
class Solution {
    public int minSetSize(int[] arr) {
        // Find the maximum value in arr to determine the range of counts.
        int maxValue = 0;
        for (int num : arr) {
            maxValue = Math.max(maxValue, num);
        }

        // Create and populate a frequency array where index represents the number from arr
        // and the value at that index represents the count of that number in arr.
        int[] frequency = new int[maxValue + 1];
        for (int num : arr) {
            ++frequency[num];
        }

        // Sort the frequency array to have the most frequent numbers at the end.
        Arrays.sort(frequency);

        // Initialize variables to track the number of elements chosen and their cumulative count.
        int setSize = 0;
        int removedElementsCount = 0;

        // Iterate from the end of the frequency array towards the beginning,
        // adding the count of each number to our cumulative count and incrementing setSize.
        // Stop once we've removed enough elements to reduce arr to half its size.
        for (int i = maxValue; i >= 0; --i) {
            if (frequency[i] > 0) {
                removedElementsCount += frequency[i];
                ++setSize;
                // If the removed elements count is at least half the size of arr, return setSize.
                if (removedElementsCount * 2 >= arr.length) {
                    return setSize;
                }
            }
        }

        // The loop should always return before reaching this point;
        // hence we don't need a return statement here, but the function requires a return type.
        // So ideally we could throw an exception or ensure the loop condition always holds true.
        // However, as per the constraints and correct functionality, this line will never be reached.
        return -1; // Placeholder return statement (should not be reached).
    }
}

```

C++ Solution

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

class Solution {
public:
    int minSetSize(vector<int>& arr) {
        // Find the maximum value in the array to define the range of counts
        int maxVal = *max_element(arr.begin(), arr.end());

        // Create a frequency array to count occurrences of each element
        // Initialize it with zero using `vector` instead of plain array
        vector<int> frequency(maxVal + 1, 0);

        // Count the frequency of each element in the array
        for (int num : arr) {
            ++frequency[num];
        }

        // Sort the frequency array in descending order to prioritize removing frequent elements
        sort(frequency.begin(), frequency.end(), greater<int>());

        // Initialize the result variable and a counter for removed elements
        int result = 0;
        int removedCount = 0;

        // Iterate over the sorted frequencies
        for (int freq : frequency) {
            // Only consider non-zero frequencies
            if (freq > 0) {
                // Increase the counter by the current frequency
                removedCount += freq;
                // Increment the result for each selected element
                ++result;

                // If the removed elements count is at least half of the total array size, stop
                if (removedCount * 2 >= arr.size()) {
                    break;
                }
            }
        }

        // Return the minimum size of the set of elements that can be removed
        return result;
    }
};

```

Typescript Solution

```typescript
function minSetSize(arr: number[]): number {
    // Map to count the frequency of each element in the array
    const frequencyCounter = new Map<number, number>();

    // Iterate through the array and populate the frequencyCounter
    for (const value of arr) {
        frequencyCounter.set(value, (frequencyCounter.get(value) ?? 0) + 1);
    }

    // Convert map values into an array and sort it in descending order
    const frequencies = Array.from(frequencyCounter.values());
    frequencies.sort((a, b) => b - a); // Sort by descending frequency

    let setSize = 0; // Initialize the minimum set size required
    let elementsCount = 0; // To track the number of elements included

    // Iterate through the sorted frequencies
    for (const count of frequencies) {
        elementsCount += count; // Add the frequency count to our total
        setSize++; // Increment the count of the set size

        // Check if we've reached or exceeded half of the array length
        if (elementsCount * 2 >= arr.length) {
            break; // We've reached the minimum set size
        }
    }

    return setSize; // Return the minimum set size
}

```

Time and Space Complexity

Time Complexity

The time complexity of the provided code can be broken down into the following operations:

1. **Creating the Counter**: The `Counter(arr)` operation has a time complexity of `O(N)` where `N` is the length of the array. It goes through each element exactly once.

2. **Sorting**: The `most_common()` method on the counter returns elements in decreasing frequency, which implies an internal sort. The complexity of this sorting is `O(K log K)` where `K` is the number of unique elements in the array.

3. **Iterating through sorted frequencies**: In the worst case, this can iterate up to `K` times. However, since it stops once the removed elements count is at least half of the array size, and considering the frequency of the elements, in practice, it might be less than `K`. Nevertheless, we consider the worst case, so this step has a complexity of `O(K)`.

When combined, the time complexity is dominated by the time complexity of the sorting step, resulting in a total time complexity of `O(N) + O(K log K) + O(K)`, which simplifies to `O(N + K log K)` since `K ≤ N`.

Space Complexity

The space complexity can be analyzed as follows:

1. **Counter**: Storing the frequency of each element in the array requires `O(K)` space, where `K` is the number of unique elements.

2. **Most Common List**: The list of tuples returned by `most_common()` also takes up `O(K)` space.

Combined, the overall space complexity of the algorithm is `O(K)`, but since `K` can be at most `N` (when all elements are unique), the space complexity simplifies to `O(N)`.