229. Majority Element II

Medium Array Hash Table Counting Sorting

Problem Description

The given problem requires determining all the elements in an integer array of size n that appear more than $\lfloor n/3 \rfloor$ times. This threshold implies that we need to find elements which are present in significant quantities, enough to surpass a third of the array's total size. The output should be a list of these dominant elements, and it is implied that the list could include zero, one, or two elements since it's not possible to have more than two elements that each appear more than $\lfloor n/3 \rfloor$ times in the same array.

Intuition

The solution approach stems from a voting algorithm, specifically an extension of the Boyer-Moore Majority Voting algorithm. The original algorithm is designed to find a majority element in an array which appears more than half of the time. In this extended problem, we are looking for elements which appear more than a third of the time, so the array can have at most 2 such elements.

Here is the thinking process broken down:

1. We initiate two potential candidates for the majority element, $e1$ and $e2$, with placeholders that don't match any elements in the array to start with. These candidates represent the two elements we think might be appearing more than $\lfloor n/3 \rfloor$ times.
2. Similarly, we have two counters, $c1$ and $c2$, that help us keep track of the number of times we have encountered $e1$ and $e2$ in the array.
3. We iterate through the array ($nums$). For each element (n), we do the following:
   - If n is equal to one of our current candidates ($e1$ or $e2$), we increment the respective counter ($c1$ or $c2$).
   - If one of the counters is at zero, it means the respective candidate is not a valid majority element or has been 'voted out', so we replace it with the current element n and set its counter to 1.
   - If n is not matching either candidate and both counters $c1$ and $c2$ are not zero, we decrement both counters. This is analogous to 'voting out' a candidate, as there must be at least three distinct numbers in the current sequence (thus each couldn't be more than $\lfloor n/3 \rfloor$ appearances).
4. Since the counters can be manipulated by elements that are not actually the desired majority elements, we make a final pass through the array to check whether our candidates truly occur more than $\lfloor n/3 \rfloor$ times. We construct the result list by only including candidates that meet this criterion.

This intuition behind the algorithm leverages the idea that if we cancel out triples of distinct elements, any element that appears more than a third of the time will still remain.

Solution Approach

The solution code provided implements the approach discussed in the intuition section. Let's walk through the implementation.

1. We initialize two counters $n1$ and $n2$ to zero. These counters will track the number of times we 'see' each candidate.
2. Similarly, we start with two candidate variables $m1$ and $m2$. Values don't matter as long as they're different from each other, hence they're initialized to 0 and 1 respectively.
3. Then, we iterate over all the elements n of the input array $nums$ and apply the Boyer-Moore Voting algorithm, extended to find elements occurring more than $\lfloor n/3 \rfloor$ times:
   - If n is equal to one of our candidates ($m1$ or $m2$), we increase the corresponding counter ($n1$ or $n2$).
   - If n is not equal to either candidate and one of the counters is zero, we set that candidate to n and reset its counter to 1. This signifies that we are considering a new element as a potential majority candidate.
   - If n is different from both of the existing candidates and both counters are not zero, we decrease both counters. This represents a 'vote out' scenario where we discard a 'set' of three different elements.
4. After the loop, we have two candidates $m1$ and $m2$. However, these are only potential majority elements because their counters may have been falsely increased by elements that are not actual majorities.
5. To ensure that our candidates are valid, we perform a final pass through the array:
   - We build a new list by including the candidates that actually occur more than $\lfloor n/3 \rfloor$ times by using the $.count()$ method.
6. The final result consists of this list, which contains zero, one, or two elements occurring more than $\lfloor n/3 \rfloor$ times in the array.

The implementation effectively uses constant space, with only a few integer variables, and linear time complexity, since we make a constant number of passes over the array. The voting algorithm is an elegant solution for such majority element problems, and its extension allows it to be used even with different occurrence thresholds.

Example Walkthrough

Consider an example array $nums = [1, 2, 3, 1, 1, 2, 1]$. This array has a size $n = 7$, so we are looking for elements that appear more than $\lfloor 7/3 \rfloor = \lfloor 2.33 \rfloor = 2$ times.

Here's a step-by-step breakdown of how the algorithm would process this example:

1. We initiate two counters $n1$ and $n2$ and set them to zero. These will track occurrences of our candidates. We also start with two variables for the candidates $m1$ and $m2$, initializing them to 0 and 1, respectively.
2. We begin iterating over the elements of $nums$:
   - Iteration 1: $n = 1$. Neither $n1$ nor $n2$ is set to 1, and since $n1$ is zero, we update $m1$ to 1 and $n1$ to 1.
   - Iteration 2: $n = 2$. n is not equal to $m1$ or $m2$, and $n2$ is zero, so we update $m2$ to 2 and $n2$ to 1.
   - Iteration 3: $n = 3$. n matches neither $m1$ nor $m2$, and both counters are not zero, so we decrease both $n1$ and $n2$ by 1.
   - Iteration 4: $n = 1$. n matches $m1$, so we increase $n1$ to 1 ($n1$ was decreased in the last step).
   - Iteration 5: $n = 1$. n matches $m1$, so we increase $n1$ to 2.
   - Iteration 6: $n = 2$. n matches $m2$, so we increase $n2$ to 1 (it was decreased earlier).
   - Iteration 7: $n = 1$. n matches $m1$, so $n1$ becomes 3.
3. After the first pass, $n1$ and $n2$ count 3 and 1 respectively. Our candidates are $m1 = 1$ and $m2 = 2$.
4. We need to confirm our candidates' actual occurrences with a second pass:
   - $m1 = 1$ appears 3 times in $nums$, which is more than $\lfloor 7/3 \rfloor = 2$, so it stays.
   - $m2 = 2$ appears 2 times, which equals $\lfloor n/3 \rfloor$ but does not exceed it.
5. Only $m1$ passed the final verification, which means 1 appears more than $\lfloor n/3 \rfloor$ times in the array.
6. The result list contains a single element $[1]$, which is the element occurring more than $\lfloor n/3 \rfloor$ times in $nums$.

Through the approach described in the solution above, we efficiently determined the elements that satisfy the required occurrence condition with this example.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def majorityElement(self, nums: List[int]) -> List[int]:
        # Initialize two potential majority elements and their respective counters.
        candidate1, candidate2 = None, None
        count1, count2 = 0, 0

        # Perform Boyer-Moore Majority Vote algorithm
        for num in nums:
            # If the current element is equal to one of the potential candidates,
            # increment their respective counters.
            if num == candidate1:
                count1 += 1
            elif num == candidate2:
                count2 += 1
            # If one of the counters becomes zero, replace the candidate with
            # the current element and reset the counter to one.
            elif count1 == 0:
                candidate1, count1 = num, 1
            elif count2 == 0:
                candidate2, count2 = num, 1
            # If the current element is not equal to any candidate and both
            # counters are non-zero, decrement both counters.
            else:
                count1 -= 1
                count2 -= 1

        # Check whether the candidates are legitimate majority elements.
        # A majority element must appear more than len(nums) / 3 times.
        return [m for m in {candidate1, candidate2} if nums.count(m) > len(nums) // 3]
```

Java

```java
import java.util.List;
import java.util.ArrayList;

class Solution {
    public List<Integer> majorityElement(int[] nums) {
        // Initialize the two potential majority elements and their counters
        int majority1 = 0, majority2 = 0;
        int count1 = 0, count2 = 0;

        // First pass: find the two majority element candidates
        for (int num : nums) {
            if (num == majority1) {
                // If the current element is equal to the first candidate, increment its counter
                count1++;
            } else if (num == majority2) {
                // If the current element is equal to the second candidate, increment its counter
                count2++;
            } else if (count1 == 0) {
                // If the first candidate's count is zero, select the current element as the first candidate
                majority1 = num;
                count1 = 1;
            } else if (count2 == 0) {
                // If the second candidate's count is zero, select the current element as the second candidate
                majority2 = num;
                count2 = 1;
            } else {
                // If the current element is not equal to either candidate, decrement both counters
                count1--;
                count2--;
            }
        }

        // Second pass: validate the candidates
        List<Integer> result = new ArrayList<>();
        count1 = 0;
        count2 = 0;

        // Count the actual occurrences of the candidates in the array
        for (int num : nums) {
            if (num == majority1) {
                count1++;
            } else if (num == majority2) {
                count2++;
            }
        }

        // Check if the candidates are majority elements (> n/3 occurrences)
        if (count1 > nums.length / 3) {
            result.add(majority1);
        }
        if (count2 > nums.length / 3) {
            result.add(majority2);
        }

        // Return the list of majority elements
        return result;
    }
}
```

C++

```cpp
#include <vector>
#include <algorithm> // Include algorithm header for count function

class Solution {
public:
    std::vector<int> majorityElement(std::vector<int>& nums) {
        int count1 = 0, count2 = 0;  // Initialize counters for two potential majority elements
        int candidate1 = 0, candidate2 = 1;  // Initialize helpers for two potential majority elements

        // Use Boyer-Moore Voting Algorithm to find potential majority elements
        for (int num : nums) {
            if (num == candidate1) {
                // If the current element is the same as candidate1, increment count1
                ++count1;
            } else if (num == candidate2) {
                // If the current element is the same as candidate2, increment count2
                ++count2;
            } else if (count1 == 0) {
                // If count1 is 0, replace candidate1 with the current element and reset count1 to 1
                candidate1 = num;
                count1 = 1;
            } else if (count2 == 0) {
                // If count2 is 0, replace candidate2 with the current element and reset count2 to 1
                candidate2 = num;
                count2 = 1;
            } else {
                // If the current element is not equal to either candidate and both counters are non-zero, decrement both counters
                --count1;
                --count2;
            }
        }

        std::vector<int> result; // Initialize an empty result vector
        // Check if the candidates are indeed majority elements
        if (std::count(nums.begin(), nums.end(), candidate1) > nums.size() / 3) {
            // If candidate1 occurrence is more than a third of the array length, add to the result
            result.push_back(candidate1);
        }
        if (candidate2 != candidate1 && std::count(nums.begin(), nums.end(), candidate2) > nums.size() / 3) {
            // If candidate2 is different from candidate1 and occurs more than a third of the array length, add to the result
            // Check candidate2 != candidate1 to avoid counting the same element twice in case of duplicates
            result.push_back(candidate2);
        }

        return result; // Return the final result
    }
};
```

TypeScript

```typescript
function majorityElement(nums: number[]): number[] {
    // Initialize counters for two potential majority elements
    let count1: number = 0;
    let count2: number = 0;
    // Initialize placeholders for two potential majority elements
    let candidate1: number = nums[0] || 0;
    let candidate2: number = nums[1] || 1;

    // Use Boyer-Moore Voting Algorithm to find potential majority elements
    for (let num of nums) {
        if (num === candidate1) {
            // If the current element is the same as candidate1, increment count1
            count1++;
        } else if (num === candidate2) {
            // If the current element is the same as candidate2, increment count2
            count2++;
        } else if (count1 === 0) {
            // If count1 is 0, replace candidate1 with the current element
            candidate1 = num;
            count1 = 1;
        } else if (count2 === 0) {
            // If count2 is 0, replace candidate2 with the current element
            candidate2 = num;
            count2 = 1;
        } else {
            // If the current element is not equal to either candidate and both counters are non-zero, decrement both counters
            count1--;
            count2--;
        }
    }

    // Initialize a result array to store the final majority elements
    const result: number[] = [];

    // Validate if the candidates are indeed majority elements
    count1 = nums.filter(num => num === candidate1).length;
    count2 = nums.filter(num => num === candidate2).length;

    // Add candidate1 to result if it's greater than a third of nums length
    if (count1 > Math.floor(nums.length / 3)) {
        result.push(candidate1);
    }

    // Add candidate2 to result if it's different from candidate1 and count is greater than a third of nums length
    if (candidate1 !== candidate2 && count2 > Math.floor(nums.length / 3)) {
        result.push(candidate2);
    }

    // Return the final result
    return result;
}
```

```python
from typing import List

class Solution:
    def majorityElement(self, nums: List[int]) -> List[int]:
        # Initialize two potential majority elements and their respective counters.
        candidate1, candidate2 = None, None
        count1, count2 = 0, 0

        # Perform Boyer-Moore Majority Vote algorithm
        for num in nums:
            # If the current element is equal to one of the potential candidates,
            # increment their respective counters.
            if num == candidate1:
                count1 += 1
            elif num == candidate2:
                count2 += 1
            # If one of the counters becomes zero, replace the candidate with
            # the current element and reset the counter to one.
            elif count1 == 0:
                candidate1, count1 = num, 1
            elif count2 == 0:
                candidate2, count2 = num, 1
            # If the current element is not equal to any candidate and both
            # counters are non-zero, decrement both counters.
            else:
                count1 -= 1
                count2 -= 1

        # Check whether the candidates are legitimate majority elements.
        # A majority element must appear more than len(nums) / 3 times.
        return [m for m in {candidate1, candidate2} if nums.count(m) > len(nums) // 3]
```

Time and Space Complexity

Time Complexity

The given code implements the Boyer-Moore Voting Algorithm variant to find all elements that appear more than $\frac{N}{3}$ times in the list. The algorithm is separated into two major phases:

1. **Selection Phase**: This phase runs through the list once ($O(N)$ time), maintaining two potential majority elements along with their corresponding counters.
2. **Validation Phase**: The phase ensures that the potential elements actually appear more than $\frac{N}{3}$ times. It requires another traversal through the list for each of the two candidates to count the occurrences ($O(N)$ time per candidate).

The total time complexity is $O(N)$ for the selection phase plus $2 * O(N)$ for the validation phase, resulting in $O(N)$.

Space Complexity

The space complexity is $O(1)$, as the algorithm uses a constant amount of extra space to maintain the potential majority element candidates and their counters.