2357. Make Array Zero by Subtracting Equal Amounts

Easy | Greedy | Array | Hash Table | Sorting | Simulation | Heap (Priority Queue)

Problem Description

In this problem, we are given an array `nums` containing non-negative integers. Our task is to perform a series of operations to make all elements in the array equal to zero. An operation consists of two steps:

1. Choose a positive integer `x` that is no greater than the smallest non-zero element in `nums`.
2. Subtract this `x` from every positive element in `nums`.

The goal is to find the minimum number of such operations required to reduce all elements in the array to zero.

Intuition

To approach this problem, we observe that each operation will reduce all non-zero elements by the same amount: the smallest non-zero number in the array. This implies that each unique non-zero number in the array will eventually need to become the smallest non-zero element through these operations, and be reduced to zero.

Given this, intuitively, if we repeatedly perform the operation on the current smallest non-zero element, we would effectively eliminate that element in the next step (since it will become zero). In the case of duplicate non-zero elements, these can be removed in the same operation. Therefore, the minimum number of operations needed will be equal to the number of unique non-zero values in the array.

The Python solution reflects this intuition by first filtering out all the zeros and then transforming the remaining numbers into a set, which automatically removes duplicates, leaving us with unique non-zero numbers. The length of this resultant set is exactly the number of operations needed since that's the number of unique non-zero elements we will need to reduce to zero, one by one.

Solution Approach

The implementation of the solution is quite straightforward, leveraging Python's set data structure and list comprehension.

Here's a step-by-step breakdown of the solution code:

1. `return len({x for x in nums if x})`: This line of code encapsulates the entire solution in a compact form.
   - `{x for x in nums if x}` is a set comprehension, which iterates over each element `x` in the list `nums`.
   - The `if x` part is a conditional that filters out all zero elements. This is important because the problem specifies that we should only consider positive elements for subtraction operations.
   - By using a set rather than a list, duplicate non-zero values are automatically eliminated. This is crucial to finding the unique non-zero values.
2. `len(...)`: After the set is created with only unique non-zero values, the `len` function is used to count the number of elements in this set.

No additional algorithms or complex patterns are required for this operation; the solution is primarily based on the properties of sets in Python that give us unique values naturally.

The algorithm's complexity is O(n), where n is the number of elements in the input list `nums`. This is because the set comprehension iterates over the list once, and set operations are generally O(1) on average. Therefore, the overall computation is very efficient for this problem.

Example Walkthrough

Let's consider the example array `nums` with the following integers:

```
nums = [1, 2, 0, 2, 3]
```

To follow the solution approach using the Python code mentioned earlier, we perform the following steps:

1. The set comprehension `{x for x in nums if x}` evaluates as follows:
   - Begin iterating over each element `x` in `nums`.
   - Check `if x` to filter out zeroes, leaving us with just the positive integers `[1, 2, 2, 3]`.
   - As these values are being added to a set, the duplicates are removed, resulting in the unique non-zero values: `{1, 2, 3}`.
2. Now, we apply the `len(...)` function on this resulting set `{1, 2, 3}`.
   - The `len` function counts the number of unique non-zero elements, resulting in `3`.

Therefore, the minimum number of operations required to reduce all elements in `nums` to zero is `3`. These operations would be performed as follows:

- **First operation:** Choose `x = 1`, the smallest positive integer in `nums`. Subtract 1 from all positive elements to get `[0, 1, 0, 1, 2]`.
- **Second operation:** Now the smallest non-zero element is `1`. Subtract 1 again from all positive elements to get `[0, 0, 0, 0, 1]`.
- **Third operation:** Finally, choose `x = 1` one last time to subtract from the remaining positive element. The resulting array is `[0, 0, 0, 0, 0]`.

Now all elements in the array are zero, and it took us 3 operations, which is consistent with the length of the set computed earlier.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def minimumOperations(self, nums: List[int]) -> int:
        # Create a set comprehension to filter out all non-zero unique elements.
        unique_non_zero_numbers = {number for number in nums if number}

        # The length of this set represents the minimum number of operations needed,
        # since each unique non-zero number can be reduced to zero in one operation.
        return len(unique_non_zero_numbers)

# Example usage:
# sol = Solution()
# result = sol.minimumOperations([1, 5, 0, 1, 0])
# print(result)  # Output would be 2 (for the numbers 1 and 5)
```

Java

```java
class Solution {
    public int minimumOperations(int[] nums) {
        // Initialize a boolean array to keep track of visited numbers
        boolean[] seenNumbers = new boolean[101];
        seenNumbers[0] = true; // Assuming 0 is not considered as an operation
        int operationCount = 0; // Initialize a counter for the minimum number of operations

        // Loop through each number in the input array
        for (int number : nums) {
            // If the number has not been seen before
            if (!seenNumbers[number]) {
                operationCount++; // Increment the operation count
                seenNumbers[number] = true; // Mark the number as seen
            }
        }
        // Return the count of the minimum number of operations needed
        return operationCount;
    }
}
```

C++

```cpp
#include <vector> // Required for std::vector

// Definition of the Solution class
class Solution {
public:
    // Function to find the minimum number of operations needed to make all elements in an array unique
    int minimumOperations(std::vector<int>& nums) {
        // Create an array to keep track of numbers we've seen
        bool seen[101] = {false};
        seen[0] = true; // We start by marking 0 as seen (if we're assuming that nums only contains positive integers,
                        // then this is redundant as it would never be used)

        int operations = 0; // Initialize the count of operations to 0

        // Loop through each number in the input vector
        for (int& num : nums) {
            // If we haven't seen this number before
            if (!seen[num]) {
                operations++;     // Increment the count of operations
                seen[num] = true; // Mark the number as seen
            }
        }

        // Return the total number of operations required
        return operations;
    }
};
```

TypeScript

```typescript
/**
 * This function calculates the minimum number of operations to make all elements of an array equal.
 * An operation is defined as incrementing n - 1 elements by 1.
 *
 * @param {number[]} nums - The input array of numbers.
 * @returns {number} The minimum number of operations to make all elements equal.
 */
function minimumOperations(nums: number[]): number {
    // Initialize a set to store unique non-zero elements
    const uniqueNonZeroElements = new Set<number>();

    // Iterate through the input array
    for (let num of nums) {
        // If the current number is not zero, add it to the set
        if (num !== 0) {
            uniqueNonZeroElements.add(num);
        }
    }

    // The size of the set gives the minimum number of operations
    // since we need to make only the unique non-zero elements equal
    return uniqueNonZeroElements.size;
}
```


```python
from typing import List

class Solution:
    def minimumOperations(self, nums: List[int]) -> int:
        # Create a set comprehension to filter out all non-zero unique elements.
        unique_non_zero_numbers = {number for number in nums if number}

        # The length of this set represents the minimum number of operations needed,
        # since each unique non-zero number can be reduced to zero in one operation.
        return len(unique_non_zero_numbers)

# Example usage:
# sol = Solution()
# result = sol.minimumOperations([1, 5, 0, 1, 0])
# print(result)  # Output would be 2 (for the numbers 1 and 5)
```


Time and Space Complexity

The given Python code defines a method called `minimumOperations` that calculates the minimum number of operations needed to make all elements in the array `nums` equal to zero, under the assumption that in one operation, you can choose any non-zero element and reduce it to zero. This is inferred by the requirement to count unique non-zero elements, as setting each unique non-zero number to zero is effectively the operation implied.

Time Complexity

The time complexity of the code is dominated by the comprehension `{x for x in nums if x}` which iterates through each element of the `nums` list once. The membership check for sets in Python is `O(1)` on average. Therefore, the overall time complexity of creating this set is `O(n)`, where `n` is the length of the `nums` list.

Space Complexity

The space complexity is influenced by the additional set that is being created to store the unique non-zero elements. In the worst case, if all elements are unique and non-zero, the set will grow to the same size as the input list. Thus, the space complexity is `O(n)`, where `n` is the size of the input list. If the input list has many zeros or duplicate elements, the space used will be less than `n`.