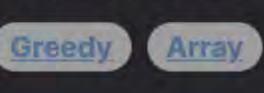
2598. Smallest Missing Non-negative Integer After Operations Leetcode Link



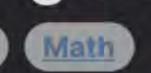












Problem Description

This problem presents an optimization challenge with a combination of array manipulation and a novel concept called MEX (minimum excluded), which represents the smallest non-negative integer not present in the array. We are given an array nums and an integer value. The operation at our disposal allows us to add or subtract value from any element in nums. The goal is to apply this operation as many times as we want to maximize the MEX of the resulting array.

Understanding the MEX is crucial as it is the key evaluation metric for the outcome of array manipulations. For instance, in an array [-1, 2, 3], since 0 and 1 are not present, the MEX is 0, as it is the smallest non-negative integer not in the array. However, if we had [1, 0, 3], the MEX would be 2. What makes this problem interesting is the strategic decision of how to add or subtract 'value' in order to maximize the MEX.

To solve the problem, we need a smart way to track the potential MEX without exhaustively enumerating all possible permutations of

Intuition

affect the remainder when each element of nums is divided by value. Consequently, the remainders form certain 'buckets' that we can only fill up to a certain level - dictated by how many times a remainder appears in the original array. This leads to the insight that we can use the frequencies of these remainders to determine the MEX without iterating through all possible array operations. Specifically, by initializing a counter for the frequency of each remainder (modulo value) present in the

array changes, which would be computationally infeasible. The intuition here lies in realizing that adding or subtracting value doesn't

array, we can iterate from 0 and move up to find the smallest non-negative integer (our potential MEX) that doesn't collide with existing remainder frequencies. When the counter for a remainder corresponding to i % value is 0, it signals that we've hit a 'gap', or a value that can't be created through any combination of operations on the current array, since none of the array elements produce a remainder equal to 1 %

value when divided by value. That's our desired maximum MEX. If not, we decrement the count and move to the next integer, repeating the process until we find our MEX. **Solution Approach**

The solution strategy is built on a counting approach, which significantly simplifies the operation and determination of MEX. Let's

expound on the implementation of the solution: 1. Counting Modulo Frequencies: We realize that the operation of adding or subtracting value from any element in nums retains its

each iteration:

modulo value. Thus, we store the count of occurrences of each remainder after dividing the elements by value using a Counter data structure. This gives us the frequency distribution of remainders.

2. Modulo Value as a Key Insight: We use the modulo operation considering value as an invariant in our problem. This insight

- solves the optimization problem by allowing us to work with a limited number of remainders (from 0 to value 1) and their counts instead of dealing with potentially large numbers after multiple addition or subtraction operations. 3. Iterating for MEX: We initialize an iteration from 0 upwards, continuing to len(nums) + 1 to cover all potential MEX values. In
- We check if the count for the current number i modulo value is 0. If it is, no number in nums can be changed through addition or subtraction operations to get a remainder that matches i % value. Hence, i represents a MEX that cannot be obtained through any variant of the nums array, and we immediately return it as the maximum MEX.
- achieve this remainder. We continue to the next iteration to find the MEX. This approach avoids brute-force computation and provides an elegant, efficient path to determining the MEX. We rely on the cyclic nature of modulo operation and the Counters that track the possibility of numbers to be fashioned into a specific modulus bucket

If the count is not 0, it implies that 1 % value is a possible remainder and thereby not the MEX. We decrement the count for

the remainder 1 % value by 1 because, conceptually, we have 'used up' one instance where a number in the array could

The algorithm is optimized as it avoids iterating through all possible operations on the array and instead, operates on a simple linear search for the smallest non-present modulo within the range, thus optimizing it to a time complexity of O(n).

Let's illustrate the solution with an example. Consider the array nums = [1, 3, 4] and let value = 3. We want to maximize the MEX of the array by adding or subtracting 3 to elements in nums.

First, we count the modulo frequencies. The remainders of numbers in nums when divided by value = 3 are:

Step 1: Counting Modulo Frequencies

through the given operation.

Example Walkthrough

So, our frequency distribution (remainder counts) is:

 Remainder 1: 2 occurrences Remainder 2: 0 occurrences

• 1 % 3 = 1

• 3 % 3 = 0

• 4 % 3 = 1

We acknowledge that we can deal with numbers in nums only in terms of their remainders when divided by value.

Step 3: Iterating for MEX

Remainder 0: 1 occurrence

Starting from 0, we check the remainder counts:

count for remainder 0.

MEX achievable is 2 by just iterating through remainders and their counts.

def findSmallestInteger(self, nums: List[int], value: int) -> int:

Iterate over the numbers from 0 to length of nums (inclusive)

// Create an array to count occurrences of each modulus result

modulo_counter = Counter(num % value for num in nums)

for i in range(len(nums) + 1):

if modulo_counter[i % value] == 0:

int[] countModulo = new int[value];

int findSmallestInteger(vector<int>& nums, int value) {

// Initialize the countArray with zeros

memset(countArray, 0, sizeof(countArray));

// Create an array to count occurrences of each modulus value

// Fill the countArray with the count of each modulus value

++countArray[(num % value + value) % value];

// Calculate the current index modulo value

// Correct negative values using modulo operation, then increment the count

// Iterate to find the smallest integer not present in nums when modulo `value`

// Decrement the count for the current index as it is now being used

for (int num : nums) {

Create a counter to keep track of the frequency of each number modulo 'value'

If there is no number i modulo 'value' in our collection, return i

// Iterate over each number in nums and increment the count for the corresponding modulus

Step 2: Utilizing Modulo Value as an Insight

• For i = 2: The remainder 2 % 3 = 2 has a count of 0. This means that we've found a gap here; we can't get a remainder of 2 by adding or subtracting 3 from any element in nums. Hence, the MEX is 2.

For i = 1: The remainder 1 % 3 = 1 has a count of 2. Again, we decrement the count for remainder 1.

Python Solution from collections import Counter from typing import List

This concludes our example. Without performing any operations on the array nums, we have efficiently determined that the maximum

• For i = 0: The remainder 0 % 3 = 0 has a count of 1 (from the number 3 in nums). This is not our MEX, so we decrement the

return i 13 # Otherwise, decrease the count of i modulo 'value' by one 14 modulo_counter[i % value] -= 1 15 16

1 class Solution { // Method to find the smallest integer that is not present in the array when modulus with value public int findSmallestInteger(int[] nums, int value) {

Java Solution

class Solution:

10

12

9

```
countModulo[(num % value + value) % value]++;
10
11
12
           // Start looking for the smallest integer that is not in the array, by checking modulus occurrences
13
           for (int i = 0; ; ++i) { // no termination condition here since we are guaranteed to find a number eventually
14
               // Use the i % value to wrap around the countModulo array
15
               // Check if the current number has a count of zero, which means it's not present in the nums array when modulus with valu
16
17
               if (countModulo[i % value] == 0) {
                   // If it's not present, this is the smallest number we are looking for
18
19
                   return i;
20
21
               // Otherwise, decrease the count and keep looking
22
               countModulo[i % value]--;
23
24
25 }
26
C++ Solution
1 #include <vector>
2 #include <cstring>
```

24 // If the count for the current index is zero, 25 // it means the integer `i` is not present in nums as mod value 26 if (countArray[modIndex] == 0) { // Return the smallest integer not present

class Solution {

int countArray[value];

for (int num : nums) {

for (int i = 0; ; ++i) {

return i;

--countArray[modIndex];

int modIndex = i % value;

public:

11

12

13

14

16

17

18

19

20

23

28

29

30

31

32

```
33
34
           // The loop is designed to run indefinitely, will break internally.
35
36 };
37
Typescript Solution
   function findSmallestInteger(nums: number[], value: number): number {
       // Initialize a counter array with size 'value' and fill it with 0s.
       const count: number[] = new Array(value).fill(0);
       // Increment the count for each number in nums based on its modulo 'value'
       for (const num of nums) {
           const index = ((num % value) + value) % value; // Handles negative numbers
           count[index]++;
 9
10
       // Iterate to find the smallest integer not in 'nums' after modulo 'value'.
11
       for (let i = 0; ; ++i) {
12
           const index = i % value; // Calculate the index in the count array
14
           // If the count at the current index is 0, this is the smallest integer not found in 'nums'
15
           if (count[index] === 0) {
16
               return i;
20
           // Otherwise, decrement the count at the current index
           count[index]--;
21
       // The loop is intended to run indefinitely, as it will always return inside the loop body.
23
24 }
25
Time and Space Complexity
```

iteratively checking for the smallest index that is not present in the frequency counter.

Time Complexity

1. Initializing the counter with (x % value for x in nums) has a linear runtime proportional to the size of nums. 2. The subsequent for loop runs for n + 1 iterations in the worst case, as it stops as soon as it finds an integer that is not in the counter. Each operation within the for loop (accessing cnt and decrementing the count) is 0(1) thanks to Python's Counter

The given Python code defines a method findSmallestInteger meant to find the smallest integer that is not present in the input list

nums when considering each number modulo value. It manages this by creating a frequency counter for the modulo results and then

The time complexity of the function is O(n), where n is the length of the input list nums. This arises from the following operations:

- which is a hash map under the hood. Therefore, as the initialization of the counter and the for loop are both linear in terms of n and are not nested, the overall time
- complexity remains linear or O(n). Space Complexity
- The space complexity of the algorithm is O(value), where value is the input parameter to the method. This complexity is attributed to the following points:

1. The counter cnt stores at most value different keys, since each number in nums is taken modulo value, which results in a possible range of [0, value).

2. No other data structures are used that depend on the size of the input or value. Thus, the space required by the counter is directly proportional to the value, leading to a space complexity of O(value).