1558. Minimum Numbers of Function Calls to Make Target Array

```
Medium
          <u>Greedy</u>
                      Bit Manipulation
                                        Array
```

Problem Description

You are provided with two integer arrays. The first one is called nums, and it contains some integers. The second one is called arr,

To achieve your goal, you are allowed to use a special modify function, which can perform one of two operations:

1. Increment by 1: Choose any index i and increase the value of arr[i] by 1. 2. Double all: Double the value of each element in the array arr.

which is the same length as nums but initially filled with 0s. Your goal is to make arr look exactly like nums.

Your task is to figure out the minimum number of calls to the modify function required to transform arr into nums. The end goal is

to achieve this transformation with the smallest possible number of operations. Intuition

times, and then incrementing. To minimize calls to modify, we should maximize the use of the doubling operation, because each doubling operation is equivalent to many increment operations. Thus, the strategy is to:

than incrementing or doubling. The insight is that any number in nums is formed by starting with 0, potentially doubling several

The solution strategy revolves around working backwards from nums to arr, as decreasing or halving elements is easier to track

1. Find the total number of increment operations needed for each number in nums by counting the number of set bits (since each set bit represents an increment operation that has occurred).

2. Find the maximum number of doubling operations needed. This is achieved by determining the highest power of 2 required to reach the maximum number in nums, which is equivalent to the bit length of the maximum number minus 1 (since starting from 0 and doubling does not

By combining the count of set bits across all numbers with the maximum bit length, we arrive at the minimum number of operations needed to transform arr into nums.

Solution Approach

The solution uses simple bitwise operations to find the answer. Here's how the implementation goes by breaking it down into logical steps: Iterate through each number in the given nums array to calculate the total number of increment operations for each number

and the highest number of doubling operations for the whole array.

For each number, we use the bitwise operation bit_count(), which returns the count of set bits (1-bits) in the number. The

count as an operation).

count of set bits actually represents how many times the increment operation has been performed to reach this number from 0 because each increment operation can only set a bit from 0 to 1. To get the maximum number of doubling operations across all numbers in the array, we look for the maximum value in nums.

We call bit_length() on this number, which gives us the position of the highest set bit, i.e., the number of times we can potentially divide the number by 2 until it reaches 0. This represents the doubling operations needed to reach this number

from 1. Since we start with 0, not 1, we subtract 1 from the bit length as the initial state does not count as a doubling

operation. The total minimum number of function calls is the sum of all increment operations (sum(v.bit_count() for v in nums)) and the maximum number of doubling operations $(\max(0, \max(nums).bit_length() - 1))$. We also ensure that the number of

The Python implementation provided uses a list comprehension to calculate the total count of bits for all numbers in nums and

doubling operations is not negative (which can happen if nums contains all zeros), by taking max(0, ...).

finds the maximum bit length among all numbers. Then it adds these two values to get the final answer.

sum(v.bit_count() for v in nums) - This part goes through every number in nums and accumulates the total increment operations required by summing up the number of set bits in each of the numbers. 2. max(nums).bit_length() - This part finds the maximum number in nums and then calculates the length of the bits needed to

represent that number, which corresponds to the number of doubling operations needed plus one (since index starts at zero).

Subtracting 1 from the bit length - We subtract 1 because the sequence starts from 0 and doubling from 0 does not change

the number, so the first doubling operation effectively starts when going from 1 to 2. Therefore, by combining these steps, the implementation efficiently computes the minimum number of operations with a time

complexity of O(n) where n is the length of the nums array.

We want to convert the arr array: [0, 0, 0] to look exactly like nums.

Here is what the code does, broken down step by step:

- **Example Walkthrough** Let's assume we have the following input:
- Now, let's walk through the solution approach for this example: **Increment Operations:**

• For the number 3, it has a binary representation of 11, which has 2 set bits. • For the number 8, it has a binary representation of 1000, which has 1 set bit. • For the number 1, it has a binary representation of 1, which has 1 set bit.

• The maximum number in nums is 8, which has a binary representation of 1000. The bit length is 4. Therefore, we would need 3 doubling

So the total number of increment operations needed is 2 (for 3) + 1 (for 8) + 1 (for 1) = 4.

operations to get from 1 to 8. Since we start from 0, we subtract one, leaving us with 4 - 1 = 3 doubling operations.

This walk through shows that we need 7 modify operations to make arr look exactly like nums, which are [3, 8, 1], by

Now, combining both the increment and doubling operations:

Doubling Operations:

Total increment operations = 4 (from step 1)

incrementing and doubling appropriately.

double operations needed.

* An operation is defined as either:

for (int number : nums) {

* 1) Subtract 1 from an element if it's not 0 or,

// and find the maximum number simultaneously

operationsCount += __builtin_popcount(value);

operationsCount += 31 - __builtin_clz(maxNum);

let operationsCount: number = 0; // Initialize counter for minimum operations

* Calculate the minimum number of operations to make all element of nums equal to zero.

let maxNum: number = 0; // Variable to store the maximum value in nums

* @param {number[]} nums - The array of numbers to be processed.

* @return {number} The minimum number of operations required.

// Return the total count of operations needed

// If maxNum is greater than 0, add the number of bits to reach the most significant bit

// of the largest number in nums (the number of left shifts needed for the largest number)

2) Divide all elements by 2 if all elements are even

int totalOperations = 0; // to track the total minimum operations

// Calculate the sum of bit counts (number of 1's) for all numbers

int maxNumber = 0; // to track the maximum number in the array

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

highest_num = max(nums) if nums else 0

Calculate the total number of set bits (1s) in all the numbers.

This corresponds to the number of increment operations needed.

Find the maximum number in the list to determine the number of

since starting from 1 (2^0) requires no doubling operations.

We need to find the position of the highest set bit (most significant bit),

that value. This is equivalent to the bit length of the maximum number minus 1

which determines the number of times we need to double from 1 to reach

num_of_double_ops = highest_num.bit_length() - 1 if highest_num > 0 else 0

* Method to calculate minimum operations to make all elements of nums equal to zero.

num_of_increment_ops = sum(bin(num).count('1') for num in nums)

Solution Implementation

class Solution:

class Solution {

/**

nums array: [3, 8, 1]

Total doubling operations = 3 (from step 2)

Python

Therefore, the minimum number of function calls required to transform arr into nums is 4 + 3 = 7.

The total number of operations is the sum of the increment and double operations. return num_of_increment_ops + num_of_double_ops Java

```
* @param nums array of integers
* @return the minimum number of operations required
public int minOperations(int[] nums) {
```

if (maxNum) {

// Importing Array object

import { max } from 'lodash';

};

/**

TypeScript

return operationsCount;

```
maxNumber = Math.max(maxNumber, number);
            totalOperations += Integer.bitCount(number); // add number of 1's in binary representation
       // Add the length of the binary string of the maximum number minus 1
       // which accounts for the division by 2 operations
        totalOperations += Integer.toBinaryString(maxNumber).length() - 1;
        return totalOperations;
C++
#include <vector>
#include <algorithm> // For std::max
class Solution {
public:
    int minOperations(std::vector<int>& nums) {
        int operationsCount = 0; // Initialize counter for minimum operations
        int maxNum = 0; // Variable to store the maximum value in nums
       // Loop through all numbers in the vector
        for (int value : nums) {
           // Update maxNum to the maximum value encountered so far
           maxNum = std::max(maxNum, value);
           // Add the number of set bits (1s) in the current value to operationsCount
```

```
*/
  function minOperations(nums: number[]): number {
      operationsCount = 0; // Reset the counter for each function call
      maxNum = 0;
      // Loop through all numbers in the array
      for (let value of nums) {
          // Update maxNum to the maximum value encountered so far
          maxNum = max([maxNum, value]) || 0;
          // Add the number of set bits (1s) in the current value to operationsCount
          operationsCount += countSetBits(value);
      // If maxNum is greater than 0, add the number of bits to reach the most significant bit
      // (the number of left shifts needed for the largest number)
      if (maxNum) {
          operationsCount += 31 - clz(maxNum);
      // Return the total count of operations needed
      return operationsCount;
  /**
   * Count the number of set bits in an integer (Naive method).
   * @param {number} n - The number in which set bits are to be counted.
   * @return {number} The count of set bits in n.
   */
  function countSetBits(n: number): number {
      let count: number = 0;
      while (n) {
          count += n & 1; // Increment count if the least significant bit is set
          n = n >>> 1; // Right shift n by 1 bit to process the next bit
      return count;
  /**
   * Compute the number of leading zero bits in the integer's binary representation.
   * @param {number} num - The number whose leading zeros are to be counted.
   * @return {number} The count of leading zero bits in num.
  function clz(num: number): number {
      if (num === 0) return 32;
      let leadingZeros = 0;
      for (let i = 31; i >= 0; i--) {
          if ((num & (1 << i)) === 0) {</pre>
              leadingZeros++;
          } else {
              break;
      return leadingZeros;
class Solution:
   def minOperations(self, nums: List[int]) -> int:
       # Calculate the total number of set bits (1s) in all the numbers.
       # This corresponds to the number of increment operations needed.
        num_of_increment_ops = sum(bin(num).count('1') for num in nums)
       # Find the maximum number in the list to determine the number of
```

finding the maximum number to calculate its bit length. Calculating the bit count for a single number is 0(1), and we do this for every number in the list, resulting in O(n) complexity where n is the length of the list. Finding the maximum value in the list also

highest_num = max(nums) if nums else 0

return num_of_increment_ops + num_of_double_ops

We need to find the position of the highest set bit (most significant bit),

that value. This is equivalent to the bit length of the maximum number minus 1

The total number of operations is the sum of the increment and double operations.

which determines the number of times we need to double from 1 to reach

num_of_double_ops = highest_num.bit_length() - 1 if highest_num > 0 else 0

since starting from 1 (2^0) requires no doubling operations.

double operations needed.

Time and Space Complexity

Thus, the time complexity of the function is O(n) where n is the size of the nums list. As for the space complexity, since no additional significant space is used apart from a few variables to store intermediate results,

takes O(n) time. The bit_length function is also O(1) as it typically involves calculating the position of the highest bit set in the

integer representation, which does not depend on the size of the number itself but on the number of bits.

The time complexity of the code can be calculated based on two operations: the calculation of bit counts for all numbers and

• Time Complexity: 0(n) Space Complexity: 0(1)

the space complexity is 0(1).

In summary: