2449. Minimum Number of Operations to Make Arrays Similar

## **Problem Description**

**Greedy** Array

Sorting

task is to perform a specific operation as many times as needed to make the nums array similar to the target array. The operation allows you to choose two distinct indices i and j (both greater than or equal to 0 and less than the length of nums) and modify nums by increasing the value at nums [i] by 2 and decreasing the value at nums [j] by 2. An array is considered similar to another if both arrays contain the same elements with the same frequency, irrespective of their order. You need to determine the minimum number of such operations needed to make nums similar to target.

You are given two arrays nums and target, each containing positive integers, and importantly, both are of the same length. Your

Intuition

## The key to solving this problem lies in recognizing that the order of elements in the arrays doesn't matter when comparing for similarity—only the frequency of each element matters. Therefore, both nums and target arrays can be sorted to ensure that

Hard

elements with the same value are positioned at the same index. Sorting in such a way that the odd and even numbers are grouped together can simplify the comparison and the operations needed to convert nums into target. After sorting both arrays, we go through each element of nums and target in a pairwise manner. For each pair (a, b) taken from nums and target respectively at the same index, we find the absolute difference between the two. This difference represents the total amount we would need to add or subtract to make nums[i] equal to target[i]. However, since our operations change values

by 2 (adding 2 to one element and simultaneously subtracting 2 from another), the total number of operations will be the sum of all these differences divided by 4. By taking this approach, we ensure that we are counting two operations (one addition and one subtraction) for every 4 integer difference between corresponding elements of nums and target. Note that we can guarantee that nums can always be made similar to target as per the constraints given by the problem description.

**Solution Approach** 

The implementation of the solution revolves around a simple but clever approach to transforming the nums array into the target

First, we sort both nums and target arrays with a custom sorting key. The lambda function provided as the key, lambda x: (x

& 1, x), sorts the numbers first by whether they are odd or even (as determined by x & 1). By doing this, we group all the

array with the smallest number of operations as prescribed. The main algorithms and patterns used are sorting and pairwise element comparison. Here, we will walk through the steps taken in the provided solution code:

## even numbers together and all the odd numbers together within each array. This arrangement is helpful because the operation we perform—adding or subtracting 2—cannot change an odd number into an even number, and vice versa.

loop.

'blocks' of 4 the total difference adds up to.

Suppose we have the following input:

• nums array: [4, 1, 3, 7]

• target array: [5, 2, 3, 8]

The sort() method rearranges the nums and target arrays in-place, ensuring that elements of the same value within each array will align, which simplifies the comparison. The next step involves a pairwise comparison of the sorted arrays. We iterate through the arrays using zip(nums, target),

which creates pairs of elements (a, b) from corresponding indices in nums and target.

For each pair, we compute the absolute difference abs(a - b). The absolute difference tells us how far away the elements are from being equal, irrespective of which one is larger. We accumulate this difference for each pair across the entire array. Once we have the total absolute difference, we need to understand how many operations are required to reconcile this difference. Since each operation changes the overall sum of

two elements by 4 (increasing one element by 2 and decreasing another by 2), we divide the total absolute difference by 4.

This division by 4 gives us the minimum number of operations required because we are effectively measuring how many

The final return statement return sum(abs(a - b) for a, b in zip(nums, target)) // 4 neatly encapsulates the accumulation of differences and the calculation of the number of operations.

In terms of data structures, this solution uses the basic Python list and relies on built-in functions like sort() and abs() for sorting

and computation. The use of the zip() function is another efficient way to handle pairing elements from the two lists in a simple

This algorithm is very efficient because it avoids any unnecessary operations; no time is wasted trying to modify individual

elements in a sequential or single-operation manner. The sort and comparison take O(n log n) and O(n) time respectively,

resulting in a solution that is also very scalable with the array size. **Example Walkthrough** Let's walk through a simple example to illustrate the solution approach described in the problem's content.

According to our solution approach, the first step is to sort both arrays while ensuring that odd and even numbers are grouped together. We use the custom lambda function lambda x: (x & 1, x) as our sorting key, where x & 1 groups odd numbers (result 1) and even numbers (result 0) together. After sorting:

Next, we compare the sorted nums and target arrays pairwise and compute the absolute differences between corresponding

However, since each of our operations changes values by adding 2 to one element and subtracting 2 from another, we need to

Because dividing 7 by 4 gives us a remainder, we must round down to get the whole number of operations that are possible.

our example, after performing the 1 operation we can on pair (4, 8), converting it to (6, 6), we'll be left with differences that

Therefore, in this presented scenario, the solution errors in indicating that we could completely transform nums into target since

odd and even numbers cannot be interchanged, and the remaining difference of 3 cannot be fixed in integer number operations.

## elements:

• Calculated as: [4, 0, 1, 2]

find out how many sets of 4 are in the total absolute difference. We divide the total absolute difference by 4 (the amount

can't be reconciled using the defined operation because the remaining differences are not multiples of 4.

Now we sum up these absolute differences to get the total amount of difference we need to reconcile:

It's important to note that because our operation moves two points at a time (one up and one down), when the total absolute difference is not a multiple of 4, this implies there may be leftover differences that cannot be fixed with the given operation. In

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

# divide by 4 to get the minimum number of operations required

operations = sum(abs(a - b) for a, b in zip(nums, target)) // 4

List<Integer> targetEvens = new ArrayList<>();

List<Integer> targetOdds = new ArrayList<>();

# This works because an operation changes four numbers at a time.

# Sort both nums and target lists first by odd-even status and then by value.

nums.sort(key=lambda x: (x % 2, x)) # Using % instead of & for clarity.

// Distribute the numbers in nums into separate lists for evens and odds

// Distribute the numbers in target into separate lists for evens and odds

target.sort(key=lambda x: (x % 2, x)) # Using % instead of & for clarity.

# Calculate the sum of absolute differences between corresponding elements,

changed by a single operation) to find the minimum number of operations needed:

Both arrays are of length 4, which means we have four pairs of numbers to consider.

• Sorted nums array: [4, 3, 1, 7] (even numbers first, then odd numbers)

• Absolute differences: [|4-8|, |3-3|, |1-2|, |7-5|]

Thus, the minimum number of operations required is 1.

Total absolute difference: 4 + 0 + 1 + 2 = 7

Number of operations: 7 // 4 = 1

Sorted target array: [8, 3, 2, 5] (even numbers first, then odd numbers)

However, within the context of the problem constraints where such a transformation is always possible, the above approach would yield the minimum number of operations required to make nums similar to target. Solution Implementation

class Solution { public long makeSimilar(int[] nums, int[] target) { // Sort both the input arrays to make positional comparison possible Arrays.sort(nums); Arrays.sort(target);

```
// Lists to store even and odd numbers from nums array
List<Integer> numsEvens = new ArrayList<>();
List<Integer> numsOdds = new ArrayList<>();
// Lists to store even and odd numbers from target array
```

for (int value : nums) {

for (int value : target) {

if (value % 2 == 0) {

} else {

} else {

**if** (value % 2 == 0) {

numsEvens.add(value);

numsOdds.add(value);

targetEvens.add(value);

targetOdds.add(value);

return operations

**Python** 

Java

class Solution:

```
// Initialize a variable to accumulate the differences
        long totalDifference = 0;
        // Calculate the difference between corresponding even numbers
        for (int i = 0; i < numsEvens.size(); ++i) {</pre>
            totalDifference += Math.abs(numsEvens.get(i) - targetEvens.get(i));
       // Calculate the difference between corresponding odd numbers
        for (int i = 0; i < nums0dds.size(); ++i) {</pre>
            totalDifference += Math.abs(numsOdds.get(i) - targetOdds.get(i));
        // Divide the total difference by 4, as per algorithm requirement, to get the answer.
        return totalDifference / 4;
C++
class Solution {
public:
    long long makeSimilar(vector<int>& nums, vector<int>& target) {
        // Sort both the 'nums' and 'target' vectors.
        sort(nums.begin(), nums.end());
        sort(target.begin(), target.end());
        // Separate the odd and even numbers from 'nums' into 'oddsNums' and 'evensNums'.
        vector<int> oddsNums;
        vector<int> evensNums;
        // Separate the odd and even numbers from 'target' into 'oddsTarget' and 'evensTarget'.
        vector<int> oddsTarget;
        vector<int> evensTarget;
        // Distribute the odd and even numbers from 'nums' into respective vectors.
        for (int value : nums) {
            if (value & 1) { // if the number is odd
                oddsNums.emplace_back(value);
            } else {
                evensNums.emplace_back(value);
       // Distribute the odd and even numbers from 'target' into respective vectors.
        for (int value : target) {
            if (value & 1) { // if the number is odd
                oddsTarget.emplace_back(value);
            } else {
                evensTarget.emplace_back(value);
        long long totalCost = 0; // Initialize the total cost to 0.
       // Calculate the cost for making the odd parts of 'nums' and 'target' similar.
        for (size t i = 0; i < oddsNums.size(); ++i) {</pre>
            totalCost += abs(oddsNums[i] - oddsTarget[i]);
```

// Calculate the cost for making the even parts of 'nums' and 'target' similar.

// Since each swap consists of 2 increments or decrements, divide the total cost by 4.

for (size\_t i = 0; i < evensNums.size(); ++i) {</pre>

function makeSimilar(nums: number[], target: number[]): number {

// Initialize empty arrays to hold even and odd elements separately

totalDifference += Math.abs(evensFromNums[i] - evensFromTarget[i]);

totalDifference += Math.abs(oddsFromNums[i] - oddsFromTarget[i]);

// Calculate the total difference between the odd elements from nums and target

# Sort both nums and target lists first by odd-even status and then by value.

nums.sort(key=lambda x: (x % 2, x)) # Using % instead of & for clarity.

target.sort(key=lambda x: (x % 2, x)) # Using % instead of & for clarity.

# Calculate the sum of absolute differences between corresponding elements,

// Separate even and odd numbers from the original nums array

// Separate even and odd numbers from the target array

// Sort both arrays in non-decreasing order

return totalCost / 4;

nums.sort( $(a, b) \Rightarrow a - b);$ 

for (const value of nums) {

} else {

} else {

if (value % 2 === 0) {

for (const value of target) {

return totalDifference / 4;

return operations

**Space Complexity** 

Time and Space Complexity

if (value % 2 === 0) {

target.sort((a, b) => a - b);

const evensFromNums: number[] = [];

const evensFromTarget: number[] = [];

evensFromNums.push(value);

oddsFromNums.push(value);

evensFromTarget.push(value);

oddsFromTarget.push(value);

for (let i = 0; i < oddsFromNums.length; ++i) {</pre>

// Return the result as the total difference divided by 4

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

# divide by 4 to get the minimum number of operations required

# This works because an operation changes four numbers at a time.

operations = sum(abs(a - b) for a, b in zip(nums, target)) // 4

const oddsFromTarget: number[] = [];

const oddsFromNums: number[] = [];

**}**;

**TypeScript** 

totalCost += abs(evensNums[i] - evensTarget[i]);

```
// Initialize the accumulator for the total difference
let totalDifference = 0;
// Calculate the total difference between the even elements from nums and target
for (let i = 0; i < evensFromNums.length; ++i) {</pre>
```

class Solution:

```
Time Complexity
  The time complexity of the given code can be broken down into the major operations it performs:
```

- target.sort(key=lambda x: (x & 1, x)): Similarly, sorting the target list also has a time complexity of  $0(n \log n)$ . sum(abs(a - b) for a, b in zip(nums, target)): Zipping the two lists and calculating the sum of absolute differences has a
- linear time complexity, O(n), since it processes each pair of elements once. When combining these operations, the dominant factor is the sorting steps, so the overall time complexity of the makeSimilar
- function is O(n log n).

Sorting both the nums and target lists is done in-place in Python (Timsort), which has a space complexity of O(n) in the worst

nums.sort(key=lambda x: (x & 1, x)): Sorting the nums list. The sorting operation typically has a time complexity of 0(n log

case, as it may require temporary space to hold elements while sorting. The generator expression within the sum function has a space complexity of O(1), as it computes the absolute differences onthe-fly without storing them.

No additional significant space is used in the function.

n), where n is the number of elements in nums.

Therefore, the overall space complexity of the makeSimilar function is O(n).

The space complexity of the given code can be analyzed as follows: