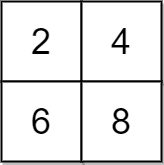
<https://leetcode.com/problems/minimum-operations-to-make-a-uni-value-grid/description/>

You are given a 2D integer grid of size m x n and an integer x. In one operation, you can **add** x to or **subtract** x from any element in the grid.

A **uni-value grid** is a grid where all the elements of it are equal.

Return *the****minimum****number of operations to make the grid****uni-value***. If it is not possible, return -1.

**Example 1:**



**Input:** grid = [[2,4],[6,8]], x = 2

**Output:** 4

**Explanation:** We can make every element equal to 4 by doing the following:

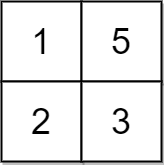
- Add x to 2 once.

- Subtract x from 6 once.

- Subtract x from 8 twice.

A total of 4 operations were used.

**Example 2:**

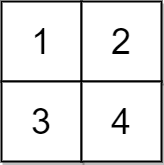


**Input:** grid = [[1,5],[2,3]], x = 1

**Output:** 5

**Explanation:** We can make every element equal to 3.

**Example 3:**



**Input:** grid = [[1,2],[3,4]], x = 2

**Output:** -1

**Explanation:** It is impossible to make every element equal.

**Constraints:**

m == grid.length

n == grid[i].length

1 <= m, n <= 10^5

1 <= m \* n <= 10^5

1 <= x, grid[i][j] <= 10^4

**Attempt 1: 2025-04-22**

**Solution 1: Math (180 min)**

**Refer to**

<https://leetcode.com/problems/minimum-operations-to-make-a-uni-value-grid/editorial/>

**Overview**

We are given a 2D integer array grid, a number x, and the ability to add or subtract x from any element in the grid any number of times. Our goal is to determine the smallest number of such operations needed to make all elements in the grid equal. If it is impossible to achieve this, we return -1.

We can see that if it is possible to make all elements equal, the optimal final value must be one of the original numbers in the grid, as any other value may require unnecessary extra steps.

For example, given grid = [[2, 4], [6, 8]] and x = 2, we can make all elements equal to 10 in 4 + 3 + 2 + 1 = 10 operations. However, this is not optimal because, along the way, we reached a state where all elements were equal to 8 in just 3 + 2 + 1 = 6 operations (not the best, but still better). From that point, increasing all numbers by 2 again is unnecessary.

**Approach 1: Sorting and Median**

**Intuition**

First, let's think about when it's possible to make all grid elements equal.

Consider any two numbers in the grid, a and b, and a number x. Suppose we want to make both a and b equal to some value v. The only operation allowed is adding or subtracting xsome number of times. This means we must be able to reachvfrom bothaandbusingx`.

For this to be possible, the differences v - a and v - b must both be multiples of x, or equivalently:

(*v*−*a*)%*x*=0and(*v*−*b*)%*x*=0

Rearranging this, we get:

*a*%*x*=*b*%*x*=*v*%*x*

This tells us that all numbers in the grid must have the same remainder when divided by x. Otherwise, it is impossible to transform them into a single value using only x-sized steps.

For example, if grid = [[1, 8], [3, 5]] and x = 2, we cannot make all elements equal to any odd value because 8 is even, and adding 2 any number of times will always result in an even number. Similarly, we cannot make all elements equal to any even value because 1, 3, and 5 are odd, and adding 2 will always keep them odd. Since we cannot make all numbers have the same parity, it is impossible to make the grid uni-value.

Thus, our first step is to check if all numbers in the grid have the same remainder when divided by x. If they don't, we immediately return -1. Otherwise, our goal is to find the smallest number of operations required.

To make things easier, note that the arrangement of numbers in the grid doesn’t affect our task at all, since we can apply operations to any number, no matter its position. So, we can simplify the problem by flattening the grid into a one-dimensional array.

Now, which value should we aim to make all numbers equal to?

If we pick a value too large, then the smaller numbers will need many additions of x to reach it.

If we pick a value too small, then the larger numbers will need many subtractions of x.

A natural choice is the **median** of the numbers.

Why? The median is the balancing point that minimizes the total distance numbers need to move. By choosing the median, we ensure that half of the numbers shift up and the other half shift down, naturally minimizing the total number of operations.

For example, consider grid = [[2, 4], [6, 8]] with x = 2:

If we make all values 8, we need 3 + 2 + 1 + 0 = 6 operations.

If we choose 4 (the median), the operations reduce to 1 + 0 + 1 + 2 = 4.

In fact, selecting the median of the numbers always results in the smallest number of operations.

The **median** value of a set of numbers is the value at which half of the numbers in the set are below it, and the other half are above it.

Let's assume that *x*=1 for simplicity. Define *f*(*i*) as the number of operations required to make all elements equal to *ai*​, where *a* is the flattened, sorted array containing all elements of the grid. Then:

*f*(*i*)=(*ai*​−*a*0​)+(*ai*​−*a*1​)+...+(*ai*​−*ai*−1​)+(*ai*+1​−*ai*​)+...+(*amn*​−*ai*​)

Similarly, for *f*(*i*−1):

*f*(*i*−1)=(*ai*−1​−*a*0​)+(*ai*−1​−*a*1​)+...+(*ai*−1​−*ai*−2​)+(*ai*​−*ai*−1​)+...+(*amn*​−*ai*−1​)

Subtracting these expressions gives:

*f*(*i*)−*f*(*i*−1)=*i*⋅(*ai*​−*ai*−1​)+(*mn*−*i*)⋅(*ai*−1​−*ai*​)=(2*i*−*mn*)(*ai*​−*ai*−1​)

Since *ai*​>*ai*−1​, the sign of *f*(*i*)−*f*(*i*−1) depends on 2*i*−*mn*:

If 2⋅*i*<*mn*, then *f*(*i*)<*f*(*i*−1), meaning that *f* is decreasing.

If 2⋅*i*>*mn*, *f*(*i*)>*f*(*i*−1), meaning that *f* is increasing.

Thus, the minimum value occurs at *f*(2*mn*​) or *f*(2*mn*−1​).

To find the median, we first sort the array in non-decreasing order and then pick the middle value. Next, we iterate through the array again to calculate how many operations are needed for each number to reach the median, and then we sum these operations.

In C++, we can avoid fully sorting the array by using the nth\_element function. This operation runs in linear time and ensures that the desired element is placed at the index it would occupy in a fully sorted array. For the median, this means the element will be placed at the middle index.

**Algorithm**

Initialize:

an empty array, called numsArray to store all numbers.

a variable result = 0 to store the total number of operations.

Flatten the grid into numsArray, by iterating over its elements and pushing them into it.

Sort numsArray in non-decreasing order.

Initialize length to the size of numsArray.

Store the median of the array (numsArray[length / 2]) in finalCommonNumber.

For each number in numsArray:

If number % x != finalCommonNumber % x, return -1, as we found two elements in the array with different remainders when divided by x.

Otherwise, increment result by the number of operations needed for this element to become equal to finalCommonNumber, i.e. abs(finalCommonNumber - number) / x.

Return result.

**Implementation**

class Solution {

public int minOperations(int[][] grid, int x) {

// Create a list to store all the numbers from the grid

ArrayList<Integer> numsArray = new ArrayList<>();

int result = 0;

// Flatten the grid into numsArray

for (int row = 0; row < grid.length; row++) {

for (int col = 0; col < grid[0].length; col++) {

numsArray.add(grid[row][col]);

}

}

// Sort numsArray in non-decreasing order to easily find the median

Collections.sort(numsArray);

int length = numsArray.size();

// Store the median element as the final common value

int finalCommonNumber = numsArray.get(length / 2);

// Iterate through each number in numsArray

for (int number : numsArray) {

// If the remainder when divided by x is different, return -1

if (number % x != finalCommonNumber % x) {

return -1;

}

// Add the number of operations required to make the current number equal to finalCommonNumber

result += Math.abs(finalCommonNumber - number) / x;

}

return result;

}

}

**Complexity Analysis**

Let *m* be the number of rows and *n* the number of columns in the grid.

Time complexity: *O*(*mn*×log*mn*)

First, we loop through the grid’s elements to flatten it into a one-dimensional array, which takes *O*(*mn*) time. Then, we sort the numsArray in *O*(*mn*×log*mn*), since it contains *m*⋅*n* elements. Finally, we go through the array, performing constant-time operations (arithmetic and checks) in each step, which takes another *O*(*mn*) time. Therefore, the overall time complexity is dominated by the sorting step and is equal to *O*(*mn*×log*mn*).

In C++, we replace sorting with the nth\_element function, which runs in *O*(2*mn*​)=*O*(*mn*) time. Therefore, the total time complexity for this implementation is equal to *O*(*mn*).

Space complexity: *O*(*mn*)

We create an array to store all numbers in the grid, which requires *O*(*mn*) space. Apart from that, we only use a fixed number of variables (finalCommonValue, result, etc.) that take up constant space.

Lastly, we must account for the space that is required for sorting (*S*), which depends on the language of implementation:

In Java, Arrays.sort() is implemented using a variant of the Quick Sort algorithm which has a space complexity of *O*(log*mn*).

In C++, the nth\_element() function has a constant space complexity of *O*(1), as it performs the rearrangement in-place without requiring additional memory proportional to the size of the input.

In Python, the sort() method sorts a list using the Timsort algorithm which is a combination of Merge Sort and Insertion Sort and has a space complexity of *O*(*mn*).

As a result, the space complexity of the algorithm is determined by the size of the numsArray and is equal to *O*(*mn*).

**Refer to Deepseek**

**Problem Understanding**

Given an m x n grid and an integer x, we need to determine the minimum number of operations to make all grid elements equal. Each operation allows us to add or subtract x from any element. If it's impossible, return -1.

**Key Insights**

**Modulo Check: All elements must have the same remainder when divided by x, otherwise it's impossible to make them equal.**

Median Property: The median minimizes the sum of absolute deviations, making it the optimal target value.

Normalization: We can work with normalized values (value / x) to simplify calculations.

**Approach**

Check Feasibility: Verify all elements modulo x give the same remainder.

Normalize Values: Convert all values to (value - base) / x where base is the remainder.

Find Median: Sort the normalized values and find the median.

Calculate Operations: Sum the absolute differences between each value and the median.

**Solution Code**

class Solution {

public int minOperations(int[][] grid, int x) {

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

int base = grid[0][0] % x;

// Collect all numbers and check modulo consistency

for (int[] row : grid) {

for (int num : row) {

if (num % x != base) {

return -1;

}

nums.add(num);

}

}

// Normalize values: (num - base) / x

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

for (int num : nums) {

normalized.add((num - base) / x);

}

Collections.sort(normalized);

int median = normalized.get(normalized.size() / 2);

// Calculate total operations

int operations = 0;

for (int num : normalized) {

operations += Math.abs(num - median);

}

return operations;

}

}

**Explanation**

Modulo Check: We first check if all elements have the same remainder when divided by x. If not, return -1 immediately.

Normalization: We subtract the base remainder and divide by x to simplify the values (e.g., if x=2 and values are [1,3,5], they become [0,1,2]).

Median Calculation: After sorting, the median gives the optimal target value that minimizes total operations.

Operations Calculation: The sum of absolute differences between each normalized value and the median gives the minimum operations needed.

**Complexity Analysis**

Time Complexity: O(mn log mn) due to sorting the normalized values.

Space Complexity: O(mn) to store the normalized values.

This approach efficiently solves the problem by leveraging modulo arithmetic and the median property to minimize operations.

**Refer to**

[L462.Minimum Moves to Equal Array Elements II (Ref.L453)](note://WEBf26377967abd88cf360cde7f55549d27)