# 题目

给你两个正整数数组 nums1 和 nums2 ，数组的长度都是 n 。

数组 nums1 和 nums2 的 绝对差值和 定义为所有 |nums1[i] - nums2[i]|（0 <= i < n）的 总和（下标从 0 开始）。

你可以选用 nums1 中的 任意一个 元素来替换 nums1 中的 至多 一个元素，以 最小化 绝对差值和。

在替换数组 nums1 中最多一个元素 之后 ，返回最小绝对差值和。因为答案可能很大，所以需要对 109 + 7 取余 后返回。

|x| 定义为：

如果 x >= 0 ，值为 x ，或者

如果 x <= 0 ，值为 -x

示例 1：

输入：nums1 = [1,7,5], nums2 = [2,3,5]

输出：3

解释：有两种可能的最优方案：

- 将第二个元素替换为第一个元素：[1,7,5] => [1,1,5] ，或者

- 将第二个元素替换为第三个元素：[1,7,5] => [1,5,5]

两种方案的绝对差值和都是 |1-2| + (|1-3| 或者 |5-3|) + |5-5| = 3

示例 2：

输入：nums1 = [2,4,6,8,10], nums2 = [2,4,6,8,10]

输出：0

解释：nums1 和 nums2 相等，所以不用替换元素。绝对差值和为 0

示例 3：

输入：nums1 = [1,10,4,4,2,7], nums2 = [9,3,5,1,7,4]

输出：20

解释：将第一个元素替换为第二个元素：[1,10,4,4,2,7] => [10,10,4,4,2,7]

绝对差值和为 |10-9| + |10-3| + |4-5| + |4-1| + |2-7| + |7-4| = 20

提示：

n == nums1.length

n == nums2.length

1 <= n <= 105

1 <= nums1[i], nums2[i] <= 105

# 分析

## 方法一：排序+二分查找

该实现围绕“最小化替换至多一个元素后的绝对差值和”展开，核心逻辑可拆解为“计算原始差值和→寻找最优替换以最大化差值减少量→计算最终结果”三步，具体如下：

1、基础准备：排序辅助数组

- 复制 nums1 得到数组 rec 并排序。

目的：后续通过二分查找快速定位 nums1 中与 nums2[i] 最接近的元素（排序是二分查找的前提），避免暴力枚举 nums1 所有元素，提升效率。

2、计算原始绝对差值和 + 寻找最大差值减少量

遍历每个下标 i（对应 nums1[i] 和 nums2[i]），同步完成两件事：

1）累加原始绝对差值和

- 计算当前位置的原始差值 diff = |nums1[i] - nums2[i]|，并累加到 sum 中（对 mod = 1e9+7 取余，避免大数溢出）。

这一步得到“不进行任何替换时的总差值”。

2）寻找最优替换以最大化差值减少量

替换的核心目标是：用 nums1 中某元素 x 替换 nums1[i]，使“原始差值 diff 与新差值 |x - nums2[i]| 的差值”最大（即 diff - |x - nums2[i]| 最大，记为 maxn）。

由于 rec 已排序，通过 lower\_bound 找到 nums2[i] 在 rec 中的插入位置 j，此时 j 附近的元素是与 nums2[i] 最接近的候选：

- 候选1：rec[j]（若 j < n）：rec中第一个 ≥ nums2[i]的元素，是“大于等于目标的最小元素”，可能与 nums2[i] 最接近。

计算减少量：diff - (rec[j] - nums2[i])（因 rec[j] ≥ nums2[i]，|rec[j] - nums2[i]| = rec[j] - nums2[i]）。

- 候选2：rec[j-1]（若 j > 0）：rec 中最后一个 < nums2[i] 的元素，是“小于目标的最大元素”，也可能与 nums2[i] 最接近。

计算减少量：diff - (nums2[i] - rec[j-1])（因 rec[j-1] < nums2[i]，|rec[j-1] - nums2[i]| = nums2[i] - rec[j-1]）。

- 用上述两个候选的减少量更新 maxn，最终 maxn 存储“替换单个元素能减少的最大差值”。

3、计算最终结果

- 最终最小差值和 =（原始差值和 sum - 最大减少量 maxn）。

由于 sum 可能小于 maxn（理论上因 maxn ≤ diff ≤ sum 不会发生，但为避免负数值），需加 mod 后再对 mod 取余，确保结果非负且符合题目要求。

代码：

class Solution {

public:

static constexpr int mod = 1'000'000'007;

int minAbsoluteSumDiff(vector<int>& nums1, vector<int>& nums2) {

vector<int> rec(nums1);

sort(rec.begin(), rec.end());

int sum = 0, maxn = 0;

int n = nums1.size();

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

int diff = abs(nums1[i] - nums2[i]);

sum = (sum + diff) % mod;

int j = lower\_bound(rec.begin(), rec.end(), nums2[i]) - rec.begin();

if (j < n) {

maxn = max(maxn, diff - (rec[j] - nums2[i]));

}

if (j > 0) {

maxn = max(maxn, diff - (nums2[i] - rec[j - 1]));

}

}

return (sum - maxn + mod) % mod;

}

};

复杂度分析

1、时间复杂度：O(n log n)

- 排序：对 rec 数组（长度 n）排序，时间复杂度为 O(n log n)，是整个算法的时间主导项。

- 遍历与二分查找：遍历数组 n 次，每次调用 lower\_bound 进行二分查找（时间 O(log n)），总时间为 O(n log n)。

- 其他操作（计算差值、更新 sum 和 maxn）均为O(1)单次操作，不影响整体复杂度。

综上，总时间复杂度为 O(n log n)。

2、空间复杂度：O(n)

- 额外创建辅助数组 rec，其长度与 nums1 一致（均为 n），用于排序和二分查找。

- 其他变量（sum、maxn、j、diff等）均为常数级空间。

综上，总空间复杂度为 O(n)。

## 方法二：排序+二分查找

要解决“替换nums1中至多一个元素以最小化绝对差值和”的问题，核心思路是先计算原始绝对差值和，再找到替换后能最大减少差值的最优位置，最终用原始和减去最大减少量得到最小结果。

解题思路

1、计算原始绝对差值和：遍历数组，累加所有|nums1[i] - nums2[i]|，得到原始总差值total。

2、寻找最优替换方案：

- 替换的核心是：对每个位置i，用nums1中某个元素x替换nums1[i]，使差值减少量diff\_reduce = |nums1[i] - nums2[i]| - |x - nums2[i]|最大（减少量越大，总差值越小）。

- 为高效找到最优x，将nums1排序并去重（排序后可通过二分查找快速定位与nums2[i]最接近的元素，去重不影响结果但减少冗余）。

- 对每个nums2[i]，通过二分查找在排序后的nums1中找到最接近nums2[i]的两个元素（小于等于nums2[i]的最大值、大于等于nums2[i]的最小值），计算替换这两个元素的减少量，记录全局最大减少量max\_reduce。

3、计算最小绝对差值和：最终结果为(total - max\_reduce) % MOD（MOD = 1e9+7，处理大数溢出）。

代码：

const int MOD = 1e9 + 7;

class Solution {

public:

int minAbsoluteSumDiff(vector<int>& nums1, vector<int>& nums2) {

int n = nums1.size();

long long total = 0; // 原始绝对差值和（用long long避免溢出）

// 1. 计算原始绝对差值和，并复制nums1用于排序

vector<int> sorted\_nums1 = nums1;

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

total += abs(nums1[i] - nums2[i]);

}

// 2. 对sorted\_nums1排序并去重（排序用于二分查找，去重减少冗余）

sort(sorted\_nums1.begin(), sorted\_nums1.end());

// 去重：保留第一个出现的元素，删除后续重复元素

sorted\_nums1.erase(unique(sorted\_nums1.begin(), sorted\_nums1.end()), sorted\_nums1.end());

int max\_reduce = 0; // 最大差值减少量

// 3. 遍历每个位置，寻找最优替换元素，计算最大减少量

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

int target = nums2[i]; // 当前nums2的元素，寻找nums1中最接近它的元素

int original\_diff = abs(nums1[i] - target); // 原始差值

// 二分查找：找到第一个 >= target 的元素位置

auto it = lower\_bound(sorted\_nums1.begin(), sorted\_nums1.end(), target);

// 候选1：it指向的元素（>= target）

if (it != sorted\_nums1.end()) {

int current\_reduce = original\_diff - abs(\*it - target);

max\_reduce = max(max\_reduce, current\_reduce);

}

// 候选2：it的前一个元素（< target，需确保it不是第一个元素）

if (it != sorted\_nums1.begin()) {

int prev = \*(prev(it));

int current\_reduce = original\_diff - abs(prev - target);

max\_reduce = max(max\_reduce, current\_reduce);

}

}

// 4. 计算最小绝对差值和，对MOD取余（处理负数情况：+MOD后再取余）

return (total - max\_reduce + MOD) % MOD;

}

};

代码解释

1、原始差值和计算：用long long存储total，避免累加过程中因元素值大（1e5）、数组长度长（1e5）导致的溢出。

2、排序与去重：sorted\_nums1是nums1的排序去重版本，排序为二分查找打基础，去重可减少二分查找的候选元素数量（不影响结果，因为重复元素替换效果相同）。

3、二分查找最优替换元素：

- lower\_bound找到第一个>= target的元素，该元素是“大于等于目标的最小元素”，可能与target最接近。

- 前一个元素（prev(it)）是“小于目标的最大元素”，同样可能是最优候选。

- 对两个候选元素分别计算替换后的减少量，更新max\_reduce。

4、结果计算：(total - max\_reduce + MOD) % MOD确保结果非负（当total < max\_reduce时，理论上不可能，但加MOD可避免负数值），并符合题目对大数取余的要求。

复杂度分析

- 时间复杂度：O(n log n)。排序nums1的时间为O(n log n)，遍历数组n次，每次二分查找时间为O(log n)（排序后数组长度为O(n)），整体复杂度由排序主导。

- 空间复杂度：O(n)。额外存储sorted\_nums1，空间与nums1长度一致。