<https://leetcode.com/problems/minimum-cost-to-make-array-equal/description/>

You are given two **0-indexed** arrays nums and cost consisting each of n **positive** integers.

You can do the following operation **any** number of times:

Increase or decrease **any** element of the array nums by 1.

The cost of doing one operation on the ith element is cost[i].

Return the **minimum** total cost such that all the elements of the array nums become **equal**.

**Example 1:**

**Input:** nums = [1,3,5,2], cost = [2,3,1,14]

**Output:** 8

**Explanation:** We can make all the elements equal to 2 in the following way:

- Increase the 0th element one time. The cost is 2.

- Decrease the 1st element one time. The cost is 3.

- Decrease the 2nd element three times. The cost is 1 + 1 + 1 = 3.

The total cost is 2 + 3 + 3 = 8.

It can be shown that we cannot make the array equal with a smaller cost.

**Example 2:**

**Input:** nums = [2,2,2,2,2], cost = [4,2,8,1,3]

**Output:** 0

**Explanation:** All the elements are already equal, so no operations are needed.

**Constraints:**

n == nums.length == cost.length

1 <= n <= 10^5

1 <= nums[i], cost[i] <= 10^6

Test cases are generated in a way that the output doesn't exceed 2^53 - 1

**Attempt 1: 2024-12-29**

**Solution 1: Binary Search (180 min)**

class Solution {

    public long minCost(int[] nums, int[] cost) {

        // Initialize the search range for the target value 'x'.

        // The smallest value in nums could be a possible 'x', so set left to 1.

        // The largest value in nums could be a possible 'x', so set right to 1,000,000

        // (the maximum constraint of nums).

        long left = 1L;

        long right = 1000000L;

        // Adjust 'left' and 'right' based on the minimum and maximum values in 'nums'.

        for (int num : nums) {

            left = Math.min(num, left);

            right = Math.max(num, right);

        }

        // Initialize the answer with the cost of transforming all elements to 1 (arbitrary start).

        long ans = findCost(nums, cost, 1);

        // Perform binary search to find the value of 'x' that minimizes the cost.

        while (left < right) {

            // Calculate the mid-point of the current search range.

            long mid = (left + right) / 2;

            // Compute the total cost for transforming all numbers to 'mid' and 'mid + 1'.

            long y1 = findCost(nums, cost, mid);

            long y2 = findCost(nums, cost, mid + 1);

            // Update the answer with the minimum cost found so far.

            ans = Math.min(y1, y2);

            // If the cost for 'mid' is smaller than for 'mid + 1', the minimum

            // lies in the range [left, mid], so adjust the right bound.

            if (y1 < y2) {

                right = mid;

            }

            // Otherwise, the minimum lies in the range [mid + 1, right], so adjust the left bound.

            else {

                left = mid + 1;

            }

        }

        // Return the minimum cost found.

        return ans;

    }

    // Helper function to calculate the total cost of transforming all numbers in `nums`

    // to a target value `x` given the associated `cost` array.

    private long findCost(int[] nums, int[] cost, long x) {

        long res = 0L; // Initialize the result (total cost) to 0

        for (int i = 0; i < nums.length; i++) {

            // Add the cost of transforming nums[i] to x

            res += Math.abs(nums[i] - x) \* cost[i];

        }

        return res;

    }

}

Time Complexity: O(nlogn)

Space Complexity: O(1)

**Explanation of Key Components:**

findCost **Function**:

This calculates the total cost of transforming all elements in nums to the value x.

The cost of transforming a single element nums[i] to x is given by cost[i]⋅∣nums[i]−x∣, which is added to the total result.

Binary Search:

The binary search operates on the range [1,1000000] or the actual minimum and maximum values in nums.

The midpoint mid is checked to determine whether the cost decreases by moving left or right. Since the function f(x) is convex, the search can efficiently narrow down to the global minimum.

Key Decisions in Binary Search:

Compare findCost(nums, cost, mid) and findCost(nums, cost, mid + 1).

If the cost at mid is less than at mid + 1, it means the global minimum lies on the left of mid, so update right = mid.

Otherwise, the global minimum lies on the right of mid, so update left = mid + 1.

Initialization:

The left and right bounds are initialized based on the minimum and maximum values of nums to ensure the search operates within the valid range.

Termination:

The loop continues until left equals right, at which point the search range has narrowed to a single value, representing the point of minimum cost.

**Complexity:**

Time Complexity:

Binary search on the range [1,max(nums)] takes O(log⁡(max(nums))).

Each iteration of binary search calls findCost, which is O(n).

Total: O(n⋅log⁡(max(nums))).

Space Complexity:

O(1), as no additional space is used beyond variables.

This combination of binary search and convex function properties ensures an efficient solution to the problem.

**Refer to**

<https://leetcode.com/problems/minimum-cost-to-make-array-equal/solutions/2734162/java-c-python-binary-search/>

**Explanation**

Assume the final equal values are x

the total cost function y = f(x) is a **convex function** (**monotonically descending to, and then ascending from, a single minimum**).

on the range of [min(A), max(A)].

To find the minimum value of f(x),

**we can binary search x by comparing f(mid) and f(mid + 1).**

**If f(mid) <= f(mid + 1), the minimum f(x) is on the left of mid, where x <= mid**

**If f(mid) >= f(mid + 1), the minimum f(x) is on the right of mid + 1, where x >= mid.**

**Repeatly doing this while left < right, until we find the minimum value and return it.**

This method is known as trinary search, if we check f(mid1) and f(mid2).

**Complexity**

Time O(nlog(a)), where a is the range of A[i]

Space O(1)

private long findCost(int[] nums, int[] cost, long x) {

long res = 0L;

for (int i = 0; i < nums.length; i++){

res += Math.abs(nums[i] - x) \* cost[i];

}

return res;

}

public long minCost(int[] nums, int[] cost) {

long left = 1L;

long right = 1000000L;

for (int num : nums) {

left = Math.min(num, left);

right = Math.max(num, right);

}

long ans = findCost(nums, cost, 1);

while (left < right) {

long mid = (left + right) / 2;

long y1 = findCost(nums, cost, mid);

long y2 = findCost(nums, cost, mid + 1);

ans = Math.min(y1, y2);

if (y1 < y2){

right = mid;

}

else{

left = mid + 1;

}

}

return ans;

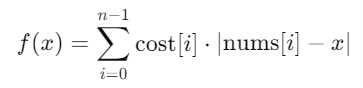
}

}

**Pure math based explanation of why Cost Function is convex**

**Refer to chatGPT**

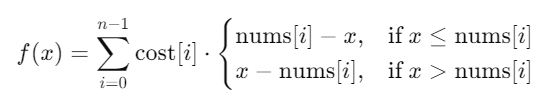
The binary search solution for LeetCode 2448 relies on the fact that the cost function f(x), defined as



is a **convex function**. This mathematical property allows us to efficiently find the global minimum of f(x) using binary search.

**Mathematical Proof of Convexity**

**1. Absolute Value Decomposition**:The function f(x) can be written as:



This splits f(x) into two linear components depending on the value of x:

For x≤nums[i], f(x) decreases linearly with slope −cost[i].

For x>nums[i], f(x) increases linearly with slope cost[i].

**2. Change in Slope**:

At x=nums[i], there is a **kink** in the function (where the slope changes).

The overall slope of f(x) is the sum of the slopes for each term in the summation.

When x is less than all values of nums[i], f(x) has a constant negative slope.

When x is greater than all values of nums[i], f(x) has a constant positive slope.

Thus, f(x) transitions from decreasing to increasing as x moves across the values of nums[i].

**3. Convexity**:

A function f(x)) is convex if its second derivative is non-negative, or equivalently, if its slope is non-decreasing. For f(x):

Between any two values of nums[i], the slope is constant (linear segments).

As x moves past nums[i], the slope increases due to the added contribution of cost[i].

Therefore, f(x) is convex because its slope is non-decreasing.

**Binary Search on Convex Function**

The binary search algorithm relies on the following key property of convex functions:

For a convex function f(x), if f(a)<f(b), then the global minimum lies in the range [a,b).

Conversely, if f(a)>f(b), the global minimum lies in the range (a,b].

In the solution:

**Midpoint Calculation**: At each step, the mid is calculated as ⌊(left+right)/2⌋.

**Cost Evaluation**: **The values f(mid) and f(mid+1) are computed:**

**If f(mid)<f(mid+1), the global minimum must be to the left, so the search range becomes [left,mid].**

**Otherwise, the global minimum is to the right, so the search range becomes [mid+1,right].**

**Convergence**: The algorithm continues until left=right, which corresponds to the point where f(x) achieves its global minimum.

**Key Insights**

**Convexity Guarantees Single Minimum**: Since f(x) is convex, it has exactly one global minimum, and binary search can efficiently locate this minimum.

**Slope Comparisons Ensure Correct Range**: The binary search steps are justified because the slope of f(x) increases monotonically. Comparing f(mid) and f(mid+1) is sufficient to decide the direction of the search.

**DP vs. W-Median vs. Binary Search**

**Refer to**

<https://leetcode.com/problems/minimum-cost-to-make-array-equal/solutions/2734091/dp-vs-w-median-vs-binary-search/>

I'd say DP is the "go to" solution for the interview.

Binary search needs some reasoning, and the weighted median is quite tricky.

**DP**

In the end, all elements of the array will be equal to some element n[i].

Well, you say, if our array is [2, 5], what if we can achieve min cost by making them 3 or 4?

For this to be the case, the cost for both 2 and 5 must be the same. But if the cost the same, we can achieve the same min cost if we pick 2 or 5.

Let's call it a pivot.

**If we sort the array, elements on the left of the pivot will increase, and on the right - decrease.**

**We go left-to-right in the sorted array, and compute cost for each pivot (cost\_l[i]).**

**Then we do the same right-to-left, compute cost (cost\_r), track and the minimum total cost (min(cost\_l[i], cost\_r)).**

long long minCost(vector<int>& n, vector<int>& cost) {

vector<long long> id(n.size()), cost\_l(n.size());

iota(begin(id), end(id), 0);

sort(begin(id), end(id), [&](int i, int j){

return n[i] < n[j];

});

for (long long i = 0, psum = 0; i < n.size() - 1; ++i) {

psum += cost[id[i]];

cost\_l[i + 1] = cost\_l[i] + psum \* (n[id[i + 1]] - n[id[i]]);

}

long long res = cost\_l.back(), cost\_r = 0;

for (long long j = n.size() - 1, psum = 0; j > 0; --j) {

psum += cost[id[j]];

cost\_r += psum \* (n[id[j]] - n[id[j - 1]]);

res = min(res, cost\_l[j - 1] + cost\_r);

}

return res;

}

**Binary Search**

This solution is based on the fact that the cost is the **convex function** (**monotonically descending to, and then ascending from, a single minimum**).

**For a given point, we compute the cost for it and its neighbor. By comparing those costs, we can tell whether the minimum is on the left or on the right.**

We binary-search for that minimum.

long long minCost(vector<int>& n, vector<int>& cost) {

long long l = 1, r = 1000000, res = 0, res1 = 0;

while (l < r) {

int m = (l + r) / 2;

res = res1 = 0;

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

res += (long long)cost[i] \* abs(n[i] - m);

res1 += (long long)cost[i] \* abs(n[i] - (m + 1));

}

if (res < res1)

r = m;

else

l = m + 1;

}

return min(res, res1);

}

**Weighted Median**

If the cost for all element is the same, then the minimum cost is when all numbers converge at the median.

Since the cost is not the same, we need to find a weighted median.

To find a weighted median, we sort elements, "repeating" each element based on its weight.

For [1,3,5,2], [2,3,1,4] case, the repeated array looks like this (median is in bold): [1,1,2,2,2,2,3,3,3,5].

We do not need to actually generate that repeated array, we can just simulate it.

We find the total weight (10), aggregate the current weight going from one side, and stop when current >= total / 2.

We sort the index array, so we do not have to combine n and cost and sort them together.

long long minCost(vector<int>& n, vector<int>& cost) {

vector<long long> id(n.size());

iota(begin(id), end(id), 0);

sort(begin(id), end(id), [&](int i, int j){ return n[i] < n[j]; });

long long total = accumulate(begin(cost), end(cost), 0LL), j = 0;

for (long long cur = 0; cur + cost[id[j]] < total / 2; ++j)

cur += cost[id[j]];

return accumulate(begin(id), end(id), 0LL, [&](long long sum, int i){

return sum + (long long)cost[id[i]] \* abs(n[id[i]] - n[id[j]]);

});

}

**Weighted Median + Binary Search**

Instead of sorting, we can use binary search to find the weigthed median.

long long minCost(vector<int>& n, vector<int>& cost) {

long long total = accumulate(begin(cost), end(cost), 0LL), res = 0;

int l = 0, r = 1000000;

while (l < r) {

long long sum = 0, m = (l + r) / 2;

for (int i = 0; i < n.size(); ++i)

sum += n[i] <= m ? cost[i] : 0;

if (sum <= total / 2)

l = m + 1;

else

r = m;

}

for (int i = 0; i < n.size(); ++i)

res += (long long)cost[i] \* abs(n[i] - l);

return res;

}

**Refer to Deepseek**

LeetCode 2448 要求我们找到使数组 nums 中所有元素相等的最小总成本。每次操作（增加或减少元素 1）的成本由 cost 数组给出。这里提供一种**基于前缀和与排序**的 Java 解法，并解释其原理。这种方法通常比二分查找更高效，且思路清晰。

**📊 Java 解法（排序与前缀和**）

import java.util.\*;

class Solution {

public long minCost(int[] nums, int[] cost) {

int n = nums.length;

int[][] pairs = new int[n][2];

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

pairs[i][0] = nums[i];

pairs[i][1] = cost[i];

}

Arrays.sort(pairs, (a, b) -> Integer.compare(a[0], b[0])); // 按 nums[i] 排序

long[] prefixWeightedSum = new long[n + 1]; // 前缀加权和 (nums[i] \* cost[i])

long[] prefixCostSum = new long[n + 1]; // 前缀成本总和 (cost[i])

// 计算前缀和数组

for (int i = 1; i <= n; i++) {

long num = pairs[i - 1][0];

long cst = pairs[i - 1][1];

prefixWeightedSum[i] = prefixWeightedSum[i - 1] + num \* cst;

prefixCostSum[i] = prefixCostSum[i - 1] + cst;

}

long minTotalCost = Long.MAX\_VALUE;

int totalElements = n;

// 考虑将每个不同的 nums[i] 作为目标值

for (int i = 1; i <= n; i++) {

long currentNum = pairs[i - 1][0];

// 计算将左侧所有元素提升到 currentNum 的成本

long leftCost = currentNum \* prefixCostSum[i - 1] - prefixWeightedSum[i - 1];

// 计算将右侧所有元素降低到 currentNum 的成本

long rightCost = (prefixWeightedSum[n] - prefixWeightedSum[i]) - currentNum \* (prefixCostSum[n] - prefixCostSum[i]);

long totalCost = leftCost + rightCost;

if (totalCost < minTotalCost) {

minTotalCost = totalCost;

}

}

return minTotalCost;

}

}

**🧠 原理解**析

**1. 问题分析**

我们需要选择一个目标值 target，使得所有 nums[i] 变为 target 的总成本最小。

总成本的计算公式为：Σ |nums[i] - target| \* cost[i]。

关键洞察：**最优的 target 一定是 nums 中已有的某个值**。这是因为成本函数 Σ |nums[i] - x| \* cost[i] 是一个分段线性函数，其最小值点必然出现在某个 nums[i] 处。

**2. 排序与前缀和**

**排序**：首先将 nums 和 cost 组成对，并按 nums 的值进行排序。这使得我们可以有序地考虑每个可能的目标值，并高效计算其左右两边的成本。

**前缀和数组**：

prefixWeightedSum[i]：存储了前 i 个元素（按排序后顺序）的 nums[j] \* cost[j] 的累积和。

prefixCostSum[i]：存储了前 i 个元素的 cost[j] 的累积和。

这两个数组帮助我们在 O(1) 时间内计算任意区间内元素的加权和或成本总和。

**3. 成本计算**

对于排序后的每个位置 i（其对应的 nums 值为 currentNum，作为候选 target），总成本分为两部分：

**左侧成本**（leftCost）：将所有小于 currentNum 的元素提升到 currentNum 的成本。

计算公式：currentNum \* prefixCostSum[i-1] - prefixWeightedSum[i-1]

解释：currentNum \* prefixCostSum[i-1] 表示如果所有左侧元素都变成 currentNum 后的理想加权和，减去它们原有的加权和 prefixWeightedSum[i-1]，即为需要填补的成本。

**右侧成本**（rightCost）：将所有大于 currentNum 的元素降低到 currentNum 的成本。

计算公式：(prefixWeightedSum[n] - prefixWeightedSum[i]) - currentNum \* (prefixCostSum[n] - prefixCostSum[i])

解释：右侧元素原有的加权和减去它们都变成 currentNum 后的理想加权和。

**总成本**：leftCost + rightCost。

**4. 复杂度**

**时间复杂度**：O(n log n)，主要由排序步骤决定。计算前缀和和遍历候选目标值都是 O(n)。

**空间复杂度**：O(n)，用于存储排序后的数组和前前缀和数组。

**📌 例子说**明

假设 nums = [1, 3, 5, 2], cost = [2, 3, 1, 14]。

**排序**：将 (nums[i], cost[i]) 按 nums[i] 排序后得到：[(1,2), (2,14), (3,3), (5,1)]。

**计算前缀和数组**：

prefixWeightedSum: [0, 1*2=2, 2+2\**14=30, 30+3\**3=39, 39+5\**1=44]

prefixCostSum: [0, 2, 2+14=16, 16+3=19, 19+1=20]

**计算每个候选目标值的成本**（以 currentNum = 2 为例，即排序后第2个元素）：

左侧成本 = 2 \* prefixCostSum[1] - prefixWeightedSum[1] = 2 \* 2 - 2 = 2

右侧成本 = (prefixWeightedSum[4] - prefixWeightedSum[2]) - 2 \* (prefixCostSum[4] - prefixCostSum[2]) = (44 - 30) - 2 \* (20 - 16) = 14 - 8 = 6

总成本 = 2 + 6 = 8

验证：将所有元素变为 2 的成本：

1 → 2: |1-2| \* 2 = 2

3 → 2: |3-2| \* 3 = 3

5 → 2: |5-2| \* 1 = 3

2 → 2: 0

总成本 = 2 + 3 + 3 = 8，与计算一致。

**💡 为什么这种方法有效**？

通过排序和前缀和，我们避免了为每个候选目标值重新遍历所有元素计算成本（O(n) per target），而是通过数学推导和预处理，将每个候选目标值的成本计算优化到 O(1)，从而将总体复杂度控制在 O(n log n)。

这种方法的优势在于其通用性和稳定性，不像二分查找那样需要担心搜索范围的边界或者函数凸性，并且其时间复杂度在有大量数据时依然表现良好。

希望这个解释对你有帮助！如果有任何疑问，欢迎继续提问。

**下面是基于二分法的介绍**

好的，我来详细解释使用**二分法**解决 LeetCode 2448 的解法。

**🔍 二分法解法原**理

二分法在这里的应用基于一个关键观察：**成本函数 f(x) = Σ |nums[i] - x| \* cost[i] 是一个凸函数（单谷函数）**，即先递减后递增，存在唯一的最小值点。

**📊 Java 解法（二分法**）

class Solution {

public long minCost(int[] nums, int[] cost) {

// 找到 nums 的最小值和最大值，确定二分搜索范围

int left = Integer.MAX\_VALUE;

int right = Integer.MIN\_VALUE;

for (int num : nums) {

left = Math.min(left, num);

right = Math.max(right, num);

}

long minCost = Long.MAX\_VALUE;

// 二分搜索找到使总成本最小的 target 值

while (left <= right) {

int mid = left + (right - left) / 2;

// 计算当前中点 mid 的总成本

long costMid = calculateCost(nums, cost, mid);

// 计算 mid + 1 的总成本，用于判断趋势

long costMidPlus = calculateCost(nums, cost, mid + 1);

// 更新最小成本

minCost = Math.min(minCost, costMid);

if (costMid < costMidPlus) {

// 成本在上升，最小值在左侧

right = mid - 1;

} else {

// 成本在下降，最小值在右侧

left = mid + 1;

}

}

return minCost;

}

// 计算将所有元素变为 target 的总成本

private long calculateCost(int[] nums, int[] cost, int target) {

long totalCost = 0;

for (int i = 0; i < nums.length; i++) {

totalCost += (long) Math.abs(nums[i] - target) \* cost[i];

}

return totalCost;

}

}

**🧠 原理解**析

**1. 成本函数的凸性**

成本函数 f(x) = Σ |nums[i] - x| \* cost[i] 具有以下性质：

当 x 很小时，函数值较大（需要增加很多元素）

随着 x 增大，函数值先减小后增大

存在一个唯一的 x 值使得函数取得最小值

这种先减后增的特性使得我们可以使用二分查找来找到最小值点。

**2. 二分搜索策略**

**搜索范围**：从 nums 数组的最小值到最大值

**判断条件**：比较 f(mid) 和 f(mid+1) 的大小关系

如果 f(mid) < f(mid+1)：最小值在 mid 左侧

如果 f(mid) > f(mid+1)：最小值在 mid 右侧

如果相等：已经找到最小值

**3. 为什么这样判断？**

因为成本函数是凸函数：

如果 f(mid) < f(mid+1)，说明函数在 mid 处已经开始上升，最小值在左侧

如果 f(mid) > f(mid+1)，说明函数在 mid 处还在下降，最小值在右侧

**📌 例子说**明

假设 nums = [1, 3, 5, 2], cost = [2, 3, 1, 14]：

**初始状态**：

left = 1, right = 5

mid = (1+5)/2 = 3

**第一次迭代**：

f(3) = |1-3|×2 + |3-3|×3 + |5-3|×1 + |2-3|×14 = 4 + 0 + 2 + 14 = 20

f(4) = |1-4|×2 + |3-4|×3 + |5-4|×1 + |2-4|×14 = 6 + 3 + 1 + 28 = 38

f(3) < f(4)，所以最小值在左侧，right = 2

**第二次迭代**：

left = 1, right = 2

mid = (1+2)/2 = 1

f(1) = |1-1|×2 + |3-1|×3 + |5-1|×1 + |2-1|×14 = 0 + 6 + 4 + 14 = 24

f(2) = |1-2|×2 + |3-2|×3 + |5-2|×1 + |2-2|×14 = 2 + 3 + 3 + 0 = 8

f(1) > f(2)，所以最小值在右侧，left = 2

**第三次迭代**：

left = 2, right = 2

找到最小值点 x = 2，最小成本为 8

**⚡ 优化版本（避免重复计算）**

class Solution {

public long minCost(int[] nums, int[] cost) {

int left = Integer.MAX\_VALUE;

int right = Integer.MIN\_VALUE;

for (int num : nums) {

left = Math.min(left, num);

right = Math.max(right, num);

}

while (left < right) {

int mid = left + (right - left) / 2;

long cost1 = calculateCost(nums, cost, mid);

long cost2 = calculateCost(nums, cost, mid + 1);

if (cost1 < cost2) {

right = mid;

} else {

left = mid + 1;

}

}

return calculateCost(nums, cost, left);

}

private long calculateCost(int[] nums, int[] cost, int target) {

long total = 0;

for (int i = 0; i < nums.length; i++) {

total += (long) Math.abs(nums[i] - target) \* cost[i];

}

return total;

}

}

**📊 复杂度分**析

**时间复杂度**：O(n log K)，其中 K 是搜索范围的大小（max - min）

每次二分搜索减少一半的搜索范围

每次计算成本需要 O(n) 时间

**空间复杂度**：O(1)

**🎯 两种方法对**比

|  |  |  |
| --- | --- | --- |
| 特性 | 前缀和法 | 二分法 |
| 时间复杂度 | O(n log n) | O(n log K) |
| 空间复杂度 | O(n) | O(1) |
| 适用场景 | 数据量适中 | 数据范围适中 |
| 优势 | 数学推导清晰 | 实现简单直观 |
| 劣势 | 需要排序和存储前缀和 | 可能重复计算成本 |

二分法的优势在于实现简单，不需要复杂的数学推导，而且空间复杂度更低。当数据范围 K = max - min 不是特别大时，二分法通常是不错的选择。

希望这个详细的解释对你有帮助！

**Refer to**

[L2968.Apply Operations to Maximize Frequency Score](note://WEB45984b791ce0bb0f0a1224f1eb84df43)