# 题目

给你两个下标从 0 开始的整数数组 nums1 和 nums2 ，长度均为 n 。

让我们定义另一个下标从 0 开始、长度为 n 的整数数组，nums3 。对于范围 [0, n - 1] 的每个下标 i ，你可以将 nums1[i] 或 nums2[i] 的值赋给 nums3[i] 。

你的任务是使用最优策略为 nums3 赋值，以最大化 nums3 中 最长非递减子数组 的长度。

以整数形式表示并返回 nums3 中 最长非递减 子数组的长度。

注意：子数组 是数组中的一个连续非空元素序列。

示例 1：

输入：nums1 = [2,3,1], nums2 = [1,2,1]

输出：2

解释：构造 nums3 的方法之一是：

nums3 = [nums1[0], nums2[1], nums2[2]] => [2,2,1]

从下标 0 开始到下标 1 结束，形成了一个长度为 2 的非递减子数组 [2,2] 。

可以证明 2 是可达到的最大长度。

示例 2：

输入：nums1 = [1,3,2,1], nums2 = [2,2,3,4]

输出：4

解释：构造 nums3 的方法之一是：

nums3 = [nums1[0], nums2[1], nums2[2], nums2[3]] => [1,2,3,4]

整个数组形成了一个长度为 4 的非递减子数组，并且是可达到的最大长度。

示例 3：

输入：nums1 = [1,1], nums2 = [2,2]

输出：2

解释：构造 nums3 的方法之一是：

nums3 = [nums1[0], nums1[1]] => [1,1]

整个数组形成了一个长度为 2 的非递减子数组，并且是可达到的最大长度。

提示：

1 <= nums1.length == nums2.length == n <= 105

1 <= nums1[i], nums2[i] <= 109

# 分析

## 方法一：动态规划

要解决“最大化nums3中最长非递减子数组长度”的问题，核心思路是动态规划，通过追踪以当前位置选择nums1或nums2时的最长非递减子数组长度，逐步更新并记录最大值。

解题思路

1、状态定义：

对于每个下标i，定义两个状态：

- dp1[i]：表示在位置i选择nums1[i]时，以i为结尾的最长非递减子数组长度。

- dp2[i]：表示在位置i选择nums2[i]时，以i为结尾的最长非递减子数组长度。

2、状态转移：

对于每个位置i（从1开始），需根据前一位置i-1的选择（nums1[i-1]或nums2[i-1]）和当前选择（nums1[i]或nums2[i]）的大小关系更新状态：

- 若选择nums1[i]：

- 若nums1[i] >= nums1[i-1]，则dp1[i]可从dp1[i-1]延续（dp1[i] = dp1[i-1] + 1）。

- 若nums1[i] >= nums2[i-1]，则dp1[i]也可从dp2[i-1]延续（取两者最大值）。

- 若均不满足，则dp1[i] = 1（仅包含当前元素）。

- 选择nums2[i]时的逻辑与dp1[i]类似，需比较nums2[i]与nums1[i-1]、nums2[i-1]的大小。

3、初始化与结果：

- 初始时，dp1[0] = 1，dp2[0] = 1（第一个元素自身构成长度为1的子数组）。

- 遍历过程中，记录dp1[i]和dp2[i]的最大值，即为最终结果。

代码：

class Solution {

public:

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

int n = nums1.size();

if (n == 0) return 0;

// 优化空间：仅用变量记录前一位置的dp1和dp2

int prev1 = 1; // 前一位置选nums1时的最长长度

int prev2 = 1; // 前一位置选nums2时的最长长度

int max\_len = 1; // 记录全局最大值

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

int curr1 = 1; // 当前位置选nums1的初始长度

int curr2 = 1; // 当前位置选nums2的初始长度

// 更新curr1：当前选nums1，看能否从前一位置的选择延续

if (nums1[i] >= nums1[i-1]) {

curr1 = max(curr1, prev1 + 1);

}

if (nums1[i] >= nums2[i-1]) {

curr1 = max(curr1, prev2 + 1);

}

// 更新curr2：当前选nums2，看能否从前一位置的选择延续

if (nums2[i] >= nums1[i-1]) {

curr2 = max(curr2, prev1 + 1);

}

if (nums2[i] >= nums2[i-1]) {

curr2 = max(curr2, prev2 + 1);

}

// 更新全局最大值

max\_len = max(max\_len, max(curr1, curr2));

// 为下一迭代更新前一位置的状态

prev1 = curr1;

prev2 = curr2;

}

return max\_len;

}

};

代码解释

1、空间优化：

由于每个位置的状态仅依赖前一位置的状态，无需存储整个dp1和dp2数组，只需用prev1、prev2记录前一位置的结果，用curr1、curr2计算当前位置的结果，空间复杂度从O(n)降至O(1)。

2、状态转移细节：

- 对于curr1（当前选nums1[i]），需检查能否从prev1（前一选nums1[i-1]）或prev2（前一选nums2[i-1]）延续，取最大可能长度。

- 对于curr2（当前选nums2[i]），逻辑同上，确保所有可能的延续路径都被考虑。

3、全局最大值更新：

每次迭代后，将当前的curr1和curr2与max\_len比较，确保记录下最长的非递减子数组长度。

复杂度分析

- 时间复杂度：O(n)，其中n是数组长度。仅需遍历数组一次，每次迭代的操作是常数级。

- 空间复杂度：O(1)，仅使用固定数量的变量存储中间结果。