# 题目

给定两个大小分别为 m 和 n 的正序（从小到大）数组 nums1 和 nums2。请你找出并返回这两个正序数组的中位数 。

算法的时间复杂度应该为 O(log (m+n)) 。

示例 1：

输入：nums1 = [1,3], nums2 = [2]

输出：2.00000

解释：合并数组 = [1,2,3] ，中位数 2

示例 2：

输入：nums1 = [1,2], nums2 = [3,4]

输出：2.50000

解释：合并数组 = [1,2,3,4] ，中位数 (2 + 3) / 2 = 2.5

提示：

nums1.length == m

nums2.length == n

0 <= m <= 1000

0 <= n <= 1000

1 <= m + n <= 2000

-106 <= nums1[i], nums2[i] <= 106

# 分析

通过二分查找的方式在两个有序数组中找到合适的分割点，使得分割点左右两边的元素满足中位数的条件。根据数组的长度奇偶性，返回相应的中位数。

这个算法的核心思想是使用二分查找的方式在两个有序数组中找到一个分割点，将数组分为两个部分，使得左侧部分的元素都小于等于右侧部分的元素。

1、初始情况：假设 nums1 的长度小于等于 nums2 的长度，确保使用二分查找时在较短的数组上进行。然后定义 left 和 right 两个指针，分别初始指向 nums1 的起始和结束位置。

2、二分查找：在每一轮的二分查找中，计算 partitionX 和 partitionY 分别表示在 nums1 和 nums2 中的分割点位置（不是中位数）。通过分割点将数组分为左右两部分。

partitionX 表示在 nums1 中的分割点位置，计算方式为 left 和 right 之间的中点。这是典型的二分查找中计算中点的方式。

partitionY 表示在 nums2 中的分割点位置。由于总的元素个数可能为奇数或偶数，这里采用 (m + n + 1) / 2 - partitionX 来计算。这个表达式的含义是通过总的元素个数减去 partitionX，从而保证 partitionY 的位置能够适应奇数和偶数的情况。

这两个分割点的计算是为了将两个数组分割为左右两部分，确保左侧部分的元素总数等于或大于右侧部分的元素总数。这样的划分是为了满足中位数的条件，从而进行二分查找找到合适的位置。在二分查找的过程中，left 和 right 的值会不断地调整，从而逼近合适的分割点位置。

3、确定分割点位置：计算分割点左右两侧的元素值，分别为 maxX、minX、maxY、minY。这里采用 INT\_MIN 和 INT\_MAX 来处理边界情况。

maxX 表示分割点左侧的最大值。如果 partitionX 为 0，说明分割点位于 nums1 的起始位置，左侧没有元素，因此将 maxX 设置为 INT\_MIN，表示负无穷。

minX 表示分割点右侧的最小值。如果 partitionX 等于 m，说明分割点位于 nums1 的结束位置，右侧没有元素，因此将 minX 设置为 INT\_MAX，表示正无穷。

这样的设定是为了处理边界情况，确保在二分查找过程中能够正确地处理分割点的左右两侧元素。在实际的二分查找中，通过这两个值，可以判断当前分割点是否满足中位数的条件，从而调整二分查找的范围。

4、判断条件：如果满足 maxX <= minY && maxY <= minX，则说明找到了合适的分割点。此时，如果数组长度之和为偶数，中位数为 (max(maxX, maxY) + min(minX, minY)) / 2.0；如果数组长度之和为奇数，中位数为 max(maxX, maxY)。

5、二分查找更新：如果 maxX > minY，说明分割点过大，需要向左移动，更新 right = partitionX - 1；如果 maxY > minX，说明分割点过小，需要向右移动，更新 left = partitionX + 1。

6、循环迭代：重复上述步骤，直到找到合适的分割点。

这个算法的关键在于通过二分查找在两个有序数组中找到一个合适的分割点，使得分割点左右两侧的元素满足中位数的条件。整个过程的时间复杂度是 O(log(min(m, n)))，其中 m 和 n 分别是两个数组的长度。

代码：

class Solution {

public:

double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {

if (nums1.size() > nums2.size()) {

swap(nums1, nums2);

}

int m = nums1.size();

int n = nums2.size();

int left = 0, right = m;

while (left <= right) {

int partitionX = (left + right) / 2;

int partitionY = (m + n + 1) / 2 - partitionX;

int maxX = (partitionX == 0) ? INT\_MIN : nums1[partitionX - 1];

int minX = (partitionX == m) ? INT\_MAX : nums1[partitionX];

int maxY = (partitionY == 0) ? INT\_MIN : nums2[partitionY - 1];

int minY = (partitionY == n) ? INT\_MAX : nums2[partitionY];

if (maxX <= minY && maxY <= minX) {

if ((m + n) % 2 == 0) {

return (max(maxX, maxY) + min(minX, minY)) / 2.0;

} else {

return max(maxX, maxY);

}

} else if (maxX > minY) {

right = partitionX - 1;

} else {

left = partitionX + 1;

}

}

return 0.0;

}

};