<https://leetcode.com/problems/median-of-two-sorted-arrays/>

Given two sorted arrays nums1 and nums2 of size m and n respectively, return **the median** of the two sorted arrays.

The overall run time complexity should be O(log (m+n)).

**Example 1:**

Input: nums1 = [1,3], nums2 = [2]

Output: 2.00000

Explanation: merged array = [1,2,3] and median is 2.

**Example 2:**

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

Output: 2.50000

Explanation: merged array = [1,2,3,4] and median is (2 + 3) / 2 = 2.5.

**Constraints:**

nums1.length == m

nums2.length == n

0 <= m <= 1000

0 <= n <= 1000

1 <= m + n <= 2000

-10^6 <= nums1[i], nums2[i] <= 10^6

**Attempt 1: 2022-09-18 (360min, too long to figure out how Binary Search solution works, this special problem requires repeat practice)**

class Solution {

    public double findMedianSortedArrays(int[] nums1, int[] nums2) {

        int len1 = nums1.length;

        int len2 = nums2.length;

        // Make sure nums1 is a shorter one

        // Refer to: Why switch two input  if(len1 > len2) { return findMedianSortedArrays(nums2, nums1); } ?

        if(len1 > len2) {

            return findMedianSortedArrays(nums2, nums1);

        }

        int lo = 0;

        // Refer to: Why we have to set 'end' equal to 'lenA' rather than 'lenA - 1' ?

        int hi = len1;

        while(lo <= hi) {

            // Based on nums1 is a shorter one, split on it first

            int split1 = lo + (hi - lo) / 2;

            // Complementary split nums2

            int split2 = (len1 + len2) / 2 - split1;

            // If split1 is 0 means nothing on the left side, assign dummy min value

            int split1\_left = split1 > 0 ? nums1[split1 - 1] : Integer.MIN\_VALUE;

            // If split1 is len1 means nothing on the right side, assign dummy max value

            int split1\_right = split1 < len1 ? nums1[split1] : Integer.MAX\_VALUE;

            // If split2 is 0 means nothing on the left side, assign dummy min value

            int split2\_left = split2 > 0 ? nums2[split2 - 1] : Integer.MIN\_VALUE;

            // If split2 is len2 means nothing on the right side, assign dummy max value

            int split2\_right = split2 < len2 ? nums2[split2] : Integer.MAX\_VALUE;

            // We have splitted array at correct position

            if(split1\_left <= split2\_right && split2\_left <= split1\_right) {

                if((len1 + len2) % 2 == 0) {

                    return ((Math.max(split1\_left, split2\_left)) + Math.min(split1\_right, split2\_right)) / 2.0;

                } else {

                    return Math.min(split1\_right, split2\_right);

                }

            // We are include more numbers on nums1 right side, shrink right side

            } else if(split1\_left > split2\_right) {

                hi = split1 - 1;

            // We are include less numbers on nums1 left side, shrink left side

            } else {

                lo = split1 + 1;

            }

        }

        return 0;

    }

}

Space Complexity: O(1)

Time Complexity: O(n^2)

**Why switch two input  if(len1 > len2) { return findMedianSortedArrays(nums2, nums1); } ?**

**Refer to**

<https://leetcode.com/problems/median-of-two-sorted-arrays/discuss/2471/Very-concise-O(log(min(MN)))-iterative-solution-with-detailed-explanation>

Since nums1 and nums2 can be mutually determined from each other, we can just move one of them first, then calculate the other accordingly. However, it is much more practical to move nums2 (the one on the shorter array) first. The reason is that on the shorter array, all positions are possible cut locations for median, but on the longer array, the positions that are too far left or right are simply impossible for a legitimate cut. For instance, [1], [2 3 4 5 6 7 8]. Clearly the cut between 2 and 3 is impossible, because the shorter array does not have that many elements to balance out the [3 4 5 6 7 8] part if you make the cut this way. Therefore, for the longer array to be used as the basis for the first cut, a range check must be performed. It would be just easier to do it on the shorter array, which requires no checks whatsoever. Also, moving only on the shorter array gives a run-time complexity of O(log(min(N1, N2)))

**Why we have to set 'end' equal to 'lenA' rather than 'lenA - 1' ?**

**Refer to**

<https://cloud.tencent.com/developer/article/1594692>

double findMedianSortedArrays(vector<int>& A, vector<int>& B) {

    int lenA = A.size();

    int lenB = B.size();

    if (lenA > lenB) {

        return findMedianSortedArrays(B, A);

    }

    int start = 0;

    int end = lenA;

    while (start <= end) {

        int partitionA = (start + end) / 2;

        int partitionB = (lenA + lenB) / 2 - partitionA;

        int leftA = (partitionA > 0) ? A[partitionA - 1] : INT\_MIN;

        int rightA = (partitionA < lenA) ? A[partitionA] : INT\_MAX;

        int leftB = (partitionB > 0) ? B[partitionB - 1] : INT\_MIN;

        int rightB = (partitionB < lenB) ? B[partitionB] : INT\_MAX;

        if (leftA <= rightB && leftB <= rightA) {

            if ((lenA + lenB) % 2) {

                return min(rightA, rightB);

            } else {

                return (max(leftA, leftB) + min(rightA, rightB)) / 2.;

            }

        } else if (leftA > rightB) {

            end = partitionA - 1;

        } else {

            start = partitionA + 1;

        }

    }

    return 0;

}

e.g

A={1,2}

B={3,4}

lenA=2

lenB=2

start=0

end=1(lenA-1)

==============================

Round 1:

start=0 <= end=1

partitionA=(0+1)/2=0

partitionB=(2+2)/2-0=2

leftA=INT\_MIN

rightA=A[0]=1

-> A: INT\_MIN | {1,2}

leftB=B[2-1]=4

rightB=INT\_MAX

-> B: {3,4} | INT\_MAX

leftB=4 > rightA=1

start=0+1=1

-----------------------

Round 2:

start=1 <= end=1

partitionA=(1+1)/2=1

partitionB=(2+2)/2-1=1

leftA=A[0]=1

rightA=A[1]=2

-> A: {1} | {2}

leftB=B[0]=3

rightB=B[1]=4

-> B: {3} | {4}

leftB=3 > rightA=2

start=1+1=2

----------------------

Now the problem for assign 'end=lenA - 1' instead of correct 'end=lenA' leading

the end of while loop beacuse start=2 > end=lenA-1=2-1=1, ideally if set correct

as end=lenA=2, start=2 <= end=2 suppose to be another Round 3.

Let's recollect different 'start' value meaning:

(1)start=0 -> split before 1st element of A, in other words means not take any

element of A, and in this case we have a dummy element as INT\_MIN before split

point, INT\_MIN | {1,2}

(2)start=1 -> split after 1st element + before 2nd element of A, {1} | {2}

(3)Then we suppose to have a split after 2nd element, in other words take all

elements of A, and in this case we have a dummy element as INT\_MAX after split

point, {1,2} | INT\_MAX, supose able to do by start=2, but as previous mentioned,

end=lenA-1=2-1=1 block while(start <= end) when start=2.

To resolve this, we have to correct 'end' assignment as 'end=lenA' instead of

'end=lenA - 1', which will enable including "split after all elements" (start=2)

scenario as case (3)

==============================

After analysis, set 'end=lenA', since start=2, end=2, the while loop while(start

<= end) will able to go through, but since 'end' changed and cause 'partitionA=

(start+end)/2' also changed, we have to derive from Round 1 again:

Round 1:

start=0 <= end=2

partitionA=(0+2)/2=1

partitionB=(2+2)/2-1=1

leftA=A[0]=1

rightA=A[1]=2

-> A: {1} | {2}

leftB=B[0]=3

rightB=B[1]=4

-> B: {3} | {4}

leftB=3 > rightA=2

start=0+1=1

-------------------------------

Round 2:

start=1 <= end=2

partitionA=(1+2)/2=1

partitionB=(2+2)/2-1=1

leftA=A[0]=1

rightA=A[1]=2

-> A: {1} | {2}

leftB=B[0]=3

rightB=B[1]=4

-> B: {3} | {4}

leftB=3 > rightA=2

start=1+1=2

-------------------------------

Round 3:

start=2 <= end=2

partitionA=(2+2)/2=2

partitionB=(2+2)/2-2=0

leftA=A[1]=2

rightA=INT\_MAX

-> A: {1,2} | INT\_MAX

leftB=INT\_MIN

rightB=B[0]=3

-> B: INT\_MIN | {3,4}

-------------------------------

It match (leftA=2 <= rightB=3 && leftB=INT\_MIN <= rightA=INT\_MAX), so we find a

solution: since (lenA + lenB) % 2 == 0, total len is even, middle is

(max(leftA, leftB) + min(rightA, rightB)) / 2

**Template refer to**

<https://cloud.tencent.com/developer/article/1594692>

**题目详情**

原题如下：

There are two sorted arrays nums1 and nums2 of size m and n respectively. Find the median of the two sorted arrays. The overall run time complexity should be O(log (m+n)).You may assume nums1 and nums2 cannot be both empty.

翻译如下：有两个有序数组 nums1和 nums2，长度分别为m和n，在最大时间复杂度为 O(log(m+n))的要求下找到将两个数组合并成的有序数组

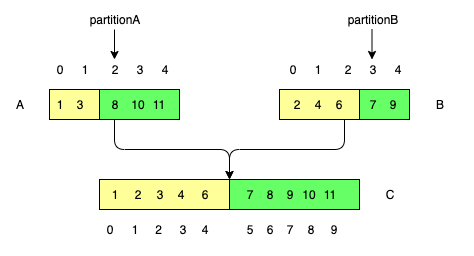
nums3的中位数。

例子：

A=[1,3,8,10,11], B=[2,4,6,7,9], 将两数组有序合并后得到 C=[1,2,3,4,6,7,8,9,10,11]，此时中位数为 (6+7)/2=6.5.

**题目分析**

**初次看到这道题时可能会毫无头绪，只知道肯定用二分查找（因为有个log)，但是并不知道应该二分求解什么对象，很多人会直奔二分求解数组 C的中位数，但是发现构造数组 C需要手动将两个数组合并，这将直接导致我们的时间复杂度达到 O(n)。并且令人头大的是当合并数组C的长度为奇数或者偶数时，中位数的求法甚至都不一样。因此，我们需要另辟蹊径。**我们先来看一下数组C有偶数个元素的情况



我们不妨如上图所示将融合的过程画出来。可以发现，数组C的左半部分（黄色）由数组A的左边一部分(左paritition)和数组B的左边一部分构成，C的右半部分(绿色)由数组A的右边一部分（右partition）和数组B的右边一部分构成。数组A被 partitionA=2分为左partition和右partition，同样，数组B被 partitionB=3分为左partition和右partition。不难发现， partitionA和 partitionB必须满足以下关系。

**partitionA + partitionB == C.Length() / 2**

**A[partitionA - 1] <= B[partitionB]**

**B[partitionB - 1] <= A[partitionA]**

**简单的说就是，A的右partition加上B的右partition必须等于C长度的一半，A左边的最大值必须小于B右边的最小值，同样B左边的最大值必须小于A右边的最小值。**

**当我们找到满足条件的这一组数时，中位数将等于：简单的说就是，取所有左partition的的最大值和所有右partition的的最小值的平均数， 6=max(3,6), 7=min(7,8), 因此中位数为 (6+7)/2=6.5。因此，只要确定 partitionA 或 partitionB 中任意一个值，我们就能推出另外一个值，并且求得中位数!**

**比方说我们可以选择求 partitionB的值。数组B的长度为5， partitionB可以是0到5的任意一个值，那么我们必须要把正确的值试出来，怎么试呢？我们可以利用数组A和B的有序性进行二分查找！以下迭代将用伪得不能再伪的伪代码来展示（注意：二分查找窗口从 [start, end] = [0,5]开始）。**

**iteration 1:**

start = 0;

end = 5;

partitionB = (start + end) / 2 = 2;

partitionA = C.Length() / 2 - 2 = 5 - 2 = 3;

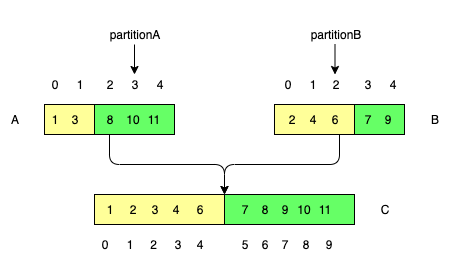
此时我们查看条件是否满足：

A[partitionA - 1] = 8

A[partitionA] = 10

B[partitionB - 1] = 4

B[partitionB] = 6



此时 A[partitionA-1]>B[partitionB], 显然不符合我们上面的不等式，值得注意的是，此时的情况是A数组的左partition的最大值大于B的右partition的最小值，因此我们的partitionB应该向右边移动，也就是说二分查找的窗口将从 [0,5] 变为 [partitionB+1,5]也就是 [3,5]。于是我们开始第二轮迭代。

**iteration 2:**

start = 3;

end = 5;

partitionB = (start + end) / 2 = 4;

partitionA = C.Length() / 2 - 4 = 5 - 4 = 1;

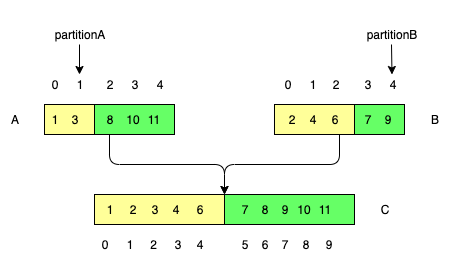
我们再次查看条件是否满足：

A[partitionA - 1] = 1

A[partitionA] = 3

B[partitionB - 1] = 7

B[partitionB] = 9



发现不等式这次仍然无法满足( B[partitionB-1]>A[partitionA])，可是此时的情况与上一轮不同，此时B数组的左partition的最大值比A数组的右partition的最小值大，因此partitionB应该向左移动。二分查找窗口将从 [3,5] 变为 [3,partitionB-1] 也就是 [3,3]。于是我们开始第三轮迭代。

**iteration 3:**

start = 3;

end = 3;

partitionB = (start + end) / 2 = 3;

partitionA = C.Length() / 2 - 2 = 5 - 3 = 2;

查看条件是否满足：

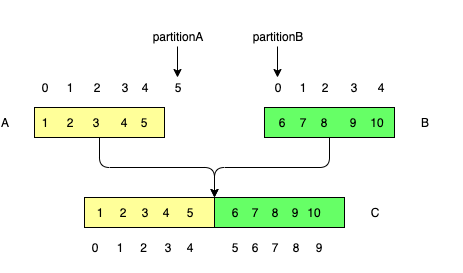
A[partitionA - 1] = 3

A[partitionA] = 8

B[partitionB - 1] = 6

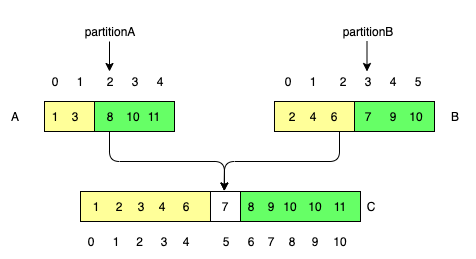
B[partitionB] = 7这时可以发现所有的左值都小于右值, 也就是说我们已经得了正确的partition！此时求得中位数为 [max(3,6)+min(8,7)]/2=6.5

在上述过程中有一个需要特别注意的边界条件，那就是 partitionB的值为5时， partitionA求得为0，这时 partitionA-1为-1，同理， partitionB的值为0时， partitionA的值为5，超过了数组的最大索引4。这种极限情况如下图所示。



**因此，当索引到数组中不存在的元素时，我们要赋值给这个元素一个特殊的值，我们规定：当 A[partitionA-1]不存在时，赋值给其 INTMIN, 因为如果A的左partition不存在，在做比较的时候任何数都可以比它里面的所有元素大。当 A[partitionA]不存在时，赋值给其 INTMAX，此时同理，如果A的右partition不存在，那么做比较的时候任何数都可以比他里面的所有元素小。对B来说同理。**

现在我们已经讨论完毕了C数组长度为偶数的情况。那么长度为奇数时如何求得中位数呢？其实答案很简单，如下图所示，只要取 partitionA和 partitionB的最小值即可。



double findMedianSortedArrays(vector<int>& A, vector<int>& B) {

    int lenA = A.size();

    int lenB = B.size();

    if (lenA > lenB) {

        return findMedianSortedArrays(B, A);

    }

    int start = 0;

    int end = lenA;

    while (start <= end) {

        int partitionA = (start + end) / 2;

        int partitionB = (lenA + lenB) / 2 - partitionA;

        int leftA = (partitionA > 0) ? A[partitionA - 1] : INT\_MIN;

        int rightA = (partitionA < lenA) ? A[partitionA] : INT\_MAX;

        int leftB = (partitionB > 0) ? B[partitionB - 1] : INT\_MIN;

        int rightB = (partitionB < lenB) ? B[partitionB] : INT\_MAX;

        if (leftA <= rightB && leftB <= rightA) {

            if ((lenA + lenB) % 2) {

                return min(rightA, rightB);

            } else {

                return (max(leftA, leftB) + min(rightA, rightB)) / 2.;

            }

        } else if (leftA > rightB) {

            end = partitionA - 1;

        } else {

            start = partitionA + 1;

        }

    }

    return 0;

}

**Similar style video Refer to**

<https://www.youtube.com/watch?v=LPFhl65R7ww>

**Another style (differ on set up hi = lenB \* 2 rather than hi = lenB) refer to:**

<https://leetcode.com/problems/median-of-two-sorted-arrays/discuss/2471/Very-concise-O(log(min(MN)))-iterative-solution-with-detailed-explanation>

This problem is notoriously hard to implement due to all the corner cases. Most implementations consider odd-lengthed and even-lengthed arrays as two different cases and treat them separately. As a matter of fact, with a little mind twist. These two cases can be combined as one, leading to a very simple solution where (almost) no special treatment is needed.

First, let's see the concept of 'MEDIAN' in a slightly unconventional way. That is:

"**if we cut the sorted array to two halves of EQUAL LENGTHS, thenmedian is the AVERAGE OF Max(lower\_half) and Min(upper\_half), i.e. thetwo numbers immediately next to the cut**".

For example, for [2 3 5 7], we make the cut between 3 and 5:

[2 3 / 5 7]

then the median = (3+5)/2. **Note that I'll use '/' to represent a cut, and (number / number) to represent a cut made through a number in this article**.

for [2 3 4 5 6], we make the cut right through 4 like this:

[2 3 (4/4) 5 7]

Since we split 4 into two halves, we say now both the lower and upper subarray contain 4. This notion also leads to the correct answer: (4 + 4) / 2 = 4;

For convenience, let's use L to represent the number immediately left to the cut, and R the right counterpart. In [2 3 5 7], for instance, we have L = 3 and R = 5, respectively.

We observe the index of L and R have the following relationship with the length of the array N:

N        Index of L / R

1              0 / 0

2              0 / 1

3              1 / 1

4              1 / 2

5              2 / 2

6              2 / 3

7              3 / 3

8              3 / 4

It is not hard to conclude that index of L = (N-1)/2, and R is at N/2. Thus, the median can be represented as(L + R)/2 = (A[(N-1)/2] + A[N/2])/2

To get ready for the two array situation, let's add a few imaginary 'positions' (represented as #'s) in between numbers, and treat numbers as 'positions' as well.

[6 9 13 18]  ->  [# 6 # 9 # 13 # 18 #]    (N = 4)

position index    0 1 2 3 4 5  6 7  8    (N\_Position = 9)

[6 9 11 13 18]->  [# 6 # 9 # 11 # 13 # 18 #]  (N = 5)

position index      0 1 2 3 4 5  6 7  8 9 10    (N\_Position = 11)

As you can see, there are always exactly 2\*N+1 'positions' regardless of length N. Therefore, the middle cut should always be made on the Nth position (0-based). Since index(L) = (N-1)/2 and index(R) = N/2 in this situation, we can infer that **index(L) = (CutPosition-1)/2, index(R) = (CutPosition)/2**.

Now for the two-array case:

A1: [# 1 # 2 # 3 # 4 # 5 #]    (N1 = 5, N1\_positions = 11)

A2: [# 1 # 1 # 1 # 1 #]    (N2 = 4, N2\_positions = 9)

Similar to the one-array problem, we need to find a cut that divides the two arrays each into two halves such that "any number in the two left halves" <= "any number in the two righthalves".

We can also make the following observations：

There are 2*N1 + 2*N2 + 2 position altogether. Therefore, there must be exactly N1 + N2 positions on each side of the cut, and 2 positions directly on the cut.

Therefore, when we cut at position C2 = K in A2, then the cut position in A1 must be C1 = N1 + N2 - k. For instance, if C2 = 2, then we must have C1 = 4 + 5 - C2 = 7.

[# 1 # 2 # 3 # (4/4) # 5 #]

[# 1 / 1 # 1 # 1 #]

When the cuts are made, we'd have two L's and two R's. They are

L1 = A1[(C1-1)/2]; R1 = A1[C1/2];

L2 = A2[(C2-1)/2]; R2 = A2[C2/2];

In the above example,

    L1 = A1[(7-1)/2] = A1[3] = 4; R1 = A1[7/2] = A1[3] = 4;

    L2 = A2[(2-1)/2] = A2[0] = 1; R2 = A1[2/2] = A1[1] = 1;

Now how do we decide if this cut is the cut we want? Because L1, L2 are the greatest numbers on the left halves and R1, R2 are the smallest numbers on the right, we only need

to make sure that any number in lower halves <= any number in upper halves. As a matter of fact, sinceL1 <= R1 and L2 <= R2 are naturally guaranteed because A1 and A2 are sorted, we only need to make sure:

L1 <= R2 and L2 <= R1.

Now we can use simple binary search to find out the result.

If we have L1 > R2, it means there are too many large numbers on the left half of A1, then we must move C1 to the left (i.e. move C2 to the right);

If L2 > R1, then there are too many large numbers on the left half of A2, and we must move C2 to the left.

Otherwise, this cut is the right one.

After we find the cut, the medium can be computed as (max(L1, L2) + min(R1, R2)) / 2;

Two side notes:

A. Since C1 and C2 can be mutually determined from each other, we can just move one of them first, then calculate the other accordingly. However, it is much more practical to move C2 (the one on the shorter array) first. The reason is that on the shorter array, all positions are possible cut locations for median, but on the longer array, the positions that are too far left or right are simply impossible for a legitimate cut. For instance, [1], [2 3 4 5 6 7 8]. Clearly the cut between 2 and 3 is impossible, because the shorter array does not have that many elements to balance out the [3 4 5 6 7 8] part if you make the cut this way. Therefore, for the longer array to be used as the basis for the first cut, a range check must be performed. It would be just easier to do it on the shorter array, which requires no checks whatsoever. Also, moving only on the shorter array gives a run-time complexity of O(log(min(N1, N2)))

B. The only edge case is when a cut falls on the 0th(first) or the 2*Nth(last) position. For instance, if C2 = 2*N2, then R2 = A2[2\*N2/2] = A2[N2], which exceeds the boundary of the array. To solve this problem, we can imagine that both A1 and A2 actually have two extra elements, INT\_MAX at A[-1] and INT\_MAX at A[N]. These additions don't change the result, but make the implementation easier: If any L falls out of the left boundary of the array, then L = INT\_MIN, and if any R falls out of the right boundary, then R = INT\_MAX.

I know that was not very easy to understand, but all the above reasoning eventually boils down to the following concise code:

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

    int N1 = nums1.size();

    int N2 = nums2.size();

    if (N1 < N2) return findMedianSortedArrays(nums2, nums1);// Make sure A2 is the shorter one.

    int lo = 0, hi = N2 \* 2;

    while (lo <= hi) {

        int mid2 = (lo + hi) / 2;  // Try Cut 2

        int mid1 = N1 + N2 - mid2;  // Calculate Cut 1 accordingly

        double L1 = (mid1 == 0) ? INT\_MIN : nums1[(mid1-1)/2];// Get L1, R1, L2, R2 respectively

        double L2 = (mid2 == 0) ? INT\_MIN : nums2[(mid2-1)/2];

        double R1 = (mid1 == N1 \* 2) ? INT\_MAX : nums1[(mid1)/2];

        double R2 = (mid2 == N2 \* 2) ? INT\_MAX : nums2[(mid2)/2];

        if (L1 > R2) lo = mid2 + 1;// A1's lower half is too big; need to move C1 left (C2 right)

        else if (L2 > R1) hi = mid2 - 1;// A2's lower half too big; need to move C2 left.

        else return (max(L1,L2) + min(R1, R2)) / 2;// Otherwise, that's the right cut.

    }

    return -1;

}