Median of Two Sorted Arrays

Problem

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.

example 1:

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

The median is 2.0

example 2:

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

The median is (2 + 3)/2 = 2.5

Solution

Approach 1: Merge two arrays

Intuition

Merge two sorted arrays into a sorted array and return the median of the merged array.

Java

```
class Solution {
    public double findMedianSortedArrays(int[] nums1, int[] nums2) {
        int p = 0, q = 0, r = 0;
        int[] newnums = new int[nums1.length + nums2.length];
        if (nums1.length == 0){
            if (nums2.length % 2 == 1)
                return nums2[(nums2.length - 1) / 2];
            else
                return (nums2[nums2.length / 2] + nums2[nums2.length / 2 - 1]) / 2.0;
        }
        else if (nums2.length == 0){
            if (nums1.length % 2 == 1)
                return nums1[(nums1.length - 1) / 2];
            else
                return (nums1[nums1.length / 2] + nums1[nums1.length / 2 - 1]) / 2.0;
        }
        while(p<nums1.length && q<nums2.length){
            if (nums1[p] < nums2[q]){
                newnums[r++] = nums1[p++];
            }
```

```
            else{
                newnums[r++] = nums2[q++];
            }
        }
        if (p<nums1.length)
            for(;p<nums1.length;p++)
                newnums[r++] = nums1[p];
        else
            for(;q<nums2.length;q++)
                newnums[r++] = nums2[q];
        r = nums1.length + nums2.length;
        if (r % 2 == 1)
            return (float) newnums[(r - 1) / 2];
        else
            return ((float) newnums[r / 2] + (float) newnums[r / 2 - 1 ]) / 2.0;
    }
}
```

Python

```
class Solution(object):
    def findMedianSortedArrays(self, nums1, nums2):
        """
        :type nums1: List[int]
        :type nums2: List[int]
        :rtype: float
        """
        l1 = len(nums1)
        l2 = len(nums2)
        if(l1 == 0):
            if (l2 % 2 == 1):
                return nums2[(l2 - 1) / 2]
            else:
                return (nums2[l2 / 2] + nums2[l2 / 2 - 1]) / 2.0
        elif(l2 == 0):
            if (l1 % 2 == 1):
                return nums1[(l1 - 1) / 2]
            else:
                return (nums1[l1 / 2] + nums1[l1 / 2 - 1]) / 2.0
        p = 0
        q = 0
        newl = []
        while(p < l1 and q < l2):
            if(nums1[p] < nums2[q]):
                newl.append(nums1[p])
                p += 1
            else:
                newl.append(nums2[q])
                q += 1
        while(p < l1):
            newl.append(nums1[p])
            p += 1
        while(q < l2):
            newl.append(nums2[q])
            q += 1
        l = l1 + l2
        if (l % 2 == 1):
            return newl[(l - 1) / 2]
        else:
            return (newl[l / 2] + newl[l / 2 - 1]) / 2.0
```

Analysis

The time complexity of such algorithm must be $O(m+n)$. I'm not sure why it could pass the test while the

requirement is $O(log(m+n))$

Approach 2: Recurrsive Approach

To solve this problem, we need to understand “What is the use of median”. In statistics, the median is used for:

> Dividing a set into two two equal length subsets, that one subset is always greater than the other.

If we undestand the use of median for diving, we are very close to the answer.
First let’s cut A into two parts at a random position i:

left_A	right_A
A[0], A[1], …, A[i-1]	A[i], A[i+1], …, A[m-1]

Since A has m elements, so there are $m+1$ kinds of cutting($i = 0...m$).
And we know:

> len(left_A) = i, len(right_A) = m − i.
> Note: when i = 0, left_A is empty, and when i = m, right_A is empty.

With the same way, cut B into two parts at a random position j
Put left_A and left_B into one set, name them left_part.

left_part	right_part
A[0], A[1], …, A[i-1]	A[i], A[i+1], …, A[m-1]
B[0], B[1], …, B[j-1]	B[j], B[j+1], …, B[n-1]

If we can ensure:

> 1. len(left_part) = len(right_part)
> 2. max(left_part) ≤ min(right_part)

then we divide all elements in {A, B} into two parts with equal length, and one part is always greater than the other. Then
$median = \frac{max(leftpart)+min(rightpart)}{2}$

To ensure these two conditions, we just need to ensure:

> 1. $i + j = m - i + n - j (or: m - i + n - j + 1)$
>   if $n \geq m, we just need to set: i = 0..m, j = \frac{m+n+1}{2} - i$
> 2. $B[j-1] \leq A[i]$ and $A[i-1] \leq B[j]$

So, all we need to do is:

> Searching i in $[0, m]$, to find an object i such that:
> $B[j-1] \leq A[i]$ and $A[i-1] \leq B[j]$ where $j = \frac{m+n+1}{2} - i$

And we can do a binary search following steps described below:
1. Set imin = 0, imax = m, then start searching in [imin, imax]
2. Set $i = \frac{imin+imax}{2}$, $j = \frac{m+n+1}{2} - i$
3. Now we have len(left_part) = len(right_part). And there are only 3 situations that we may encounter:

- $B[j-1] \leq A[i]$ and $A[i-1] \leq B[j]$
- $B[j-1] > A[i]$ (Means $A[i]$ is too small. So we must increase i.)
- $A[i-1] > B[j]$

When the object i is found, the median is:

> max$(A[i-1], B[j-1])$, when $m+n$ is odd
> $\frac{max(A[i-1],B[j-1])+min(A[i],B[j])}{2}$, when $m+n$ is even

Other explanation could be found in Solution of Median of Two Sorted Arrays

Java

```
class Solution {
    public double findMedianSortedArrays(int[] A, int[] B) {
        int m = A.length;
        int n = B.length;
        if (m > n) { // to ensure m<=n
            int[] temp = A; A = B; B = temp;
            int tmp = m; m = n; n = tmp;
        }
        int iMin = 0, iMax = m, halfLen = (m + n + 1) / 2;
        while (iMin <= iMax) {
            int i = (iMin + iMax) / 2;
            int j = halfLen - i;
            if (i < iMax && B[j-1] > A[i]){
                iMin = i + 1; // i is too small
            }
            else if (i > iMin && A[i-1] > B[j]) {
                iMax = i - 1; // i is too big
            }
            else { // i is perfect
                int maxLeft = 0;
                if (i == 0) { maxLeft = B[j-1]; }
                else if (j == 0) { maxLeft = A[i-1]; }
                else { maxLeft = Math.max(A[i-1], B[j-1]); }
                if ( (m + n) % 2 == 1 ) { return maxLeft; }

                int minRight = 0;
                if (i == m) { minRight = B[j]; }
                else if (j == n) { minRight = A[i]; }
                else { minRight = Math.min(B[j], A[i]); }

                return (maxLeft + minRight) / 2.0;
            }
        }
        return 0.0;
    }
}
```

Python

```
def median(A, B):
    m, n = len(A), len(B)
    if m > n:
        A, B, m, n = B, A, n, m
    if n == 0:
        raise ValueError

    imin, imax, half_len = 0, m, (m + n + 1) / 2
    while imin <= imax:
        i = (imin + imax) / 2
        j = half_len - i
        if i < m and B[j-1] > A[i]:
            # i is too small, must increase it
            imin = i + 1
        elif i > 0 and A[i-1] > B[j]:
            # i is too big, must decrease it
            imax = i - 1
        else:
            # i is perfect

            if i == 0: max_of_left = B[j-1]
            elif j == 0: max_of_left = A[i-1]
            else: max_of_left = max(A[i-1], B[j-1])

            if (m + n) % 2 == 1:
                return max_of_left
```