Median of Two Sorted Arrays

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)).

Solution 1

Given a sorted array A with length m, we can split it into two part:

```
{ A[0], A[1], ... , A[i - 1] } | { A[i], A[i + 1], ... , A[m - 1] }
```

All elements in right part are greater than elements in left part.

The left part has "i" elements, and right part has "m - i" elements.

There are "m + 1" kinds of splits. (i = 0 ~ m)

When i = 0, the left part has "0" elements, right part has "m" elements.

When i = m, the left part has "m" elements, right part has "0" elements.

For array B, we can split it with the same way:

```
{ B[0], B[1], ... , B[j - 1] } | { B[j], B[j + 1], ... , B[n - 1] }
```

The left part has "j" elements, and right part has "n - j" elements.

Put A's left part and B's left part into one set. (Let's name this set "LeftPart")

Put A's right part and B's right part into one set. (Let's name this set"RightPart")

```
            LeftPart           |            RightPart
{ 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) LeftPart's length == RightPart's length (or RightPart's length + 1)

 2) All elements in RightPart are greater than elements in LeftPart.
```

then we split all elements in {A, B} into two parts with eqaul length, and one part is always greater than the other part. Then the median can be easily found.

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

```
(1) i + j == m - i + n - j (or: m - i + n - j + 1)

    if n >= m, we just need to set:

           i = 0 ~ m, j = (m + n + 1) / 2 - i

(2) B[j - 1] <= A[i] and A[i - 1] <= B[j]

    considering edge values, we need to ensure:

           (j == 0 or i == m or B[j - 1] <= A[i]) and

               (i == 0 or j == n or A[i - 1] <= B[j])
```

So, all we need to do is:

```
Search i from 0 to m, to find an object "i" to meet condition (1) and (2) above.
```

And we can do this search by binary search. How?

If B[j0 - 1] > A[i0], then the object "ix" can't be in [0, i0]. Why?

```
Because if ix < i0, then jx = (m + n + 1) / 2 - ix > j0,

then B[jx - 1] >= B[j0 - 1] > A[i0] >= A[ix]. This violates

the condition (2). So ix can't be less than i0.
```

And if A[i0 - 1] > B[j0], then the object "ix" can't be in [i0, m].

So we can do the binary search following steps described below:

```
1. set imin, imax = 0, m, then start searching in [imin, imax]

2. i = (imin + imax) / 2; j = (m + n + 1) / 2 - i

3. if B[j - 1] > A[i]: continue searching in [i + 1, imax]
   elif A[i - 1] > B[j]: continue searching in [imin, i - 1]
   else: bingo! this is our object "i"
```

When the object i is found, the median is:

```
max(A[i - 1], B[j - 1]) (when m + n is odd)

or (max(A[i - 1], B[j - 1]) + min(A[i], B[j])) / 2 (when m + n is even)
```

Below is the accepted code:

```python
def median(A, B):
    m, n = len(A), len(B)

    if m > n:
        A, B, m, n = B, A, n, m

    imin, imax, half_len = 0, m, (m + n + 1) / 2
    while imin <= imax:
        i = (imin + imax) / 2
        j = half_len - i
        if j > 0 and i < m and B[j - 1] > A[i]:
            imin = i + 1
        elif i > 0 and j < n and A[i - 1] > B[j]:
            imax = i - 1
        else:
            if i == 0:
                num1 = B[j - 1]
            elif j == 0:
                num1 = A[i - 1]
            else:
                num1 = max(A[i - 1], B[j - 1])

            if (m + n) % 2 == 1:
                return num1

            if i == m:
                num2 = B[j]
            elif j == n:
                num2 = A[i]
            else:
                num2 = min(A[i], B[j])

            return (num1 + num2) / 2.0
```

written by MissMary original link here

Solution 2

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, then median is the AVERAGE OF Min(lower*half) and Max(upper*half), i.e. the two 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 right halves".

We can also make the following observations:

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

3. 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

```
L1 <= R1 && L1 <= R2 && L2 <= R1 && L2 <= R2
```

to make sure that any number in lower halves <= any number in upper halves. As a matter of fact, since L1 <= 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 > R1, 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 might as well select the shorter array (say A2) and only move C2 around, and calculate C1 accordingly. That way we can achieve 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* = 2N2, 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:

```cpp
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.

    if (N2 == 0) return ((double)nums1[(N1-1)/2] + (double)nums1[N1/2])/2;  // If A2 is empty

    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;
}
```

If you have any suggestions to make the logic and implementation even more cleaner. Please do let me know!

written by stellari original link here

Solution 3

Binary search. Call 2 times getkth and k is about half of (m + n). Every time call getkth can reduce the scale k to its half. So the time complexity is log(m + n).

```
class Solution {
public:
    int getkth(int s[], int m, int l[], int n, int k){
        // let m <= n
        if (m > n)
            return getkth(l, n, s, m, k);
        if (m == 0)
            return l[k - 1];
        if (k == 1)
            return min(s[0], l[0]);

        int i = min(m, k / 2), j = min(n, k / 2);
        if (s[i - 1] > l[j - 1])
            return getkth(s, m, l + j, n - j, k - j);
        else
            return getkth(s + i, m - i, l, n, k - i);
        return 0;
    }

    double findMedianSortedArrays(int A[], int m, int B[], int n) {
        int l = (m + n + 1) >> 1;
        int r = (m + n + 2) >> 1;
        return (getkth(A, m ,B, n, l) + getkth(A, m, B, n, r)) / 2.0;
    }
};
```

written by vaputa original link here