4. Median of Two Sorted Arrays (/problems/median-of-two-sorted-arrays/)

Aug. 30, 2017 | 55.8K views

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)).

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 Recursive Approach [Accepted]

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 equal length subsets, that one subset is always greater than the other.

If we understand the use of median for dividing, 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 ($i = 0 \sim m$).

And we know:

> $\text{len}(\text{left\_A}) = i, \text{len}(\text{right\_A}) = m - i.$
>
> 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:

```
      left_B            |        right_B
B[0], B[1], ..., B[j-1]  |  B[j], B[j+1], ..., B[n-1]
```

Put left_A and left_B into one set, and put right_A and right_B into another set. Let's name them left_part and right_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. $\text{len}(\text{left\_part}) = \text{len}(\text{right\_part})$
> 2. $\max(\text{left\_part}) \le \min(\text{right\_part})$

then we divide all elements in $\{\text{A}, \text{B}\}$ into two parts with equal length, and one part is always greater than the other. Then

$$\text{median} = \frac{\max(\text{left\_part}) + \min(\text{right\_part})}{2}$$

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

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

ps.1 For simplicity, I presume $\text{A}[i-1], \text{B}[j-1], \text{A}[i], \text{B}[j]$ are always valid even if $i = 0, i = m, j = 0$, or $j = n$. I will talk about how to deal with these edge values at last.

ps.2 Why $n \ge m$? Because I have to make sure j is non-negative since $0 \le i \le m$ and $j = \frac{m+n+1}{2} - i$. If $n < m$, then j may be negative, that will lead to wrong result.

So, all we need to do is:

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

And we can do a binary search following steps described below:

1. Set $\text{imin} = 0$, $\text{imax} = m$, then start searching in $[\text{imin}, \text{imax}]$
2. Set $i = \frac{\text{imin}+\text{imax}}{2}$, $j = \frac{m+n+1}{2} - i$
3. Now we have $\text{len}(\text{left\_part}) = \text{len}(\text{right\_part})$. And there are only 3 situations that we may encounter:
   - $\text{B}[j-1] \le \text{A}[i]$ and $\text{A}[i-1] \le \text{B}[j]$
     Means we have found the object i, so stop searching.

- $B[j-1] > A[i]$
  Means $A[i]$ is too small. We must adjust i to get $B[j-1] \leq A[i]$.
  Can we increase i?
  Yes. Because when i is increased, j will be decreased.
  So $B[j-1]$ is decreased and $A[i]$ is increased, and $B[j-1] \leq A[i]$ may be satisfied.
  Can we decrease i?
  No! Because when i is decreased, j will be increased.
  So $B[j-1]$ is increased and $A[i]$ is decreased, and $B[j-1] \leq A[i]$ will be never satisfied.
  So we must increase i. That is, we must adjust the searching range to $[i+1, \text{imax}]$.
  So, set $\text{imin} = i+1$, and goto 2.
- $A[i-1] > B[j]$:
  Means $A[i-1]$ is too big. And we must decrease i to get $A[i-1] \leq B[j]$.
  That is, we must adjust the searching range to $[\text{imin}, i-1]$.
  So, set $\text{imax} = i-1$, and goto 2.

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

Now let's consider the edges values $i=0, i=m, j=0, j=n$ where $A[i-1], B[j-1], A[i], B[j]$ may not exist. Actually this situation is easier than you think.

What we need to do is ensuring that $\max(\text{left\_part}) \leq \min(\text{right\_part})$. So, if i and j are not edges values (means $A[i-1], B[j-1], A[i], B[j]$ all exist), then we must check both $B[j-1] \leq A[i]$ and $A[i-1] \leq B[j]$. But if some of $A[i-1], B[j-1], A[i], B[j]$ don't exist, then we don't need to check one (or both) of these two conditions. For example, if $i=0$, then $A[i-1]$ doesn't exist, then we don't need to check $A[i-1] \leq B[j]$. So, what we need to do is:

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

And in a searching loop, we will encounter only three situations:

> 1. $(j=0$ or $i=m$ or $B[j-1] \leq A[i])$ and
> $(i=0$ or $j=n$ or $A[i-1] \leq B[j])$
>   Means i is perfect, we can stop searching.
> 2. $j>0$ and $i<m$ and $B[j-1] > A[i]$
>   Means i is too small, we must increase it.
> 3. $i>0$ and $j<n$ and $A[i-1] > B[j]$
>   Means i is too big, we must decrease it.

Thanks to @Quentin.chen (https://leetcode.com/Quentin.chen) for pointing out that: $i < m \implies j > 0$ and $i > 0 \implies j < n$. Because:

$$m \leq n,\ i < m \implies j = \frac{m+n+1}{2} - i > \frac{m+n+1}{2} - m \geq \frac{2m+1}{2} - m \geq 0$$

$$m \leq n,\ i > 0 \implies j = \frac{m+n+1}{2} - i < \frac{m+n+1}{2} \leq \frac{2n+1}{2} \leq n$$

So in situation 2. and 3. , we don't need to check whether $j > 0$ and whether $j < n$.

Java

Copy

```
        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 = iMin + 1; // i is too small
            }
            else if (i > iMin && A[i-1] > B[j]) {
                iMax = iMax - 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

Copy

```
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

            if i == m: min_of_right = B[j]
```

Complexity Analysis

- Time complexity: $O\big(\log\big(\min(m,n)\big)\big)$.
  At first, the searching range is $[0, m]$. And the length of this searching range will be reduced by half after each loop. So, we only need $\log(m)$ loops. Since we do constant operations in each loop, so the time complexity is $O\big(\log(m)\big)$. Since $m \leq n$, so the time complexity is $O\big(\log\big(\min(m,n)\big)\big)$.
- Space complexity: $O(1)$.
  We only need constant memory to store 9 local variables, so the space complexity is $O(1)$.

Analysis written by: @MissMary (https://leetcode.com/MissMary)

Rate this article:
(/ratings/107/232/?return=/articles/median-of-two-sorted-arrays/) (/ratings/107/232/?return=/articles/m

Articles >
Previous (/articles/binary-tree-inorder-traversal/)
Next (/articles/palindrome-number/)

Join the conversation

Login to Reply

A (https://discuss.leetcode.com/user/ankitc)

ankitC commented yesterday

You could merge the 2 sorted arrays in a manner that the end result of the merge is also sorted and that required only O(m+n) complexity. Once you have the combined array, you can take the middle element if the array length is odd or take the mean of the middle 2 elements in case of an even length array. The code seems to be long but is straightforward.

J (https://discuss.leetcode.com/user/jianqiao)

jianqiao commented 4 days ago

There is a generalized version of this problem: find the K-th largest element in M sorted arrays.

The solution is somehow complicated and is available in this paper:
Peter J. Varman, Scott D. Scheufler, Balakrishna R. Iyer, and Gary R. Ricard. 1991. Merging multiple lists on hierarchical-memory multiprocessors. J. Parallel Distrib. Comput. 12, 2 (June 1991)

The time complexity is (slightly better than) O(M log(N/M)), where N is the total size of all arrays -- note that N is typically very large and M is relatively small.

The algorithm has its practical usage -- e.g. in the partitioning phase of Parallel Multiway Merge Sort -- see the GCC libstdc++'s implementation:

https://github.com/gcc-mirror/gcc/blob/master/libstdc%2B%2B-v3/include/parallel/multiseq_selection.h#L388 (https://github.com/gcc-mirror/gcc/blob/master/libstdc%2B%2B-v3/include/parallel/multiseq_selection.h#L388)

P (https://discuss.leetcode.com/user/pvlbzn)

pvlbzn commented 4 days ago

@StefanPochmann (https://discuss.leetcode.com/uid/591) thank you, good to know! So, in this case, no $n*log(n)$ sorting bound.


(https://discuss.leetcode.com/user/stefanpochmann)

StefanPochmann commented 4 days ago

@pvlbzn (https://discuss.leetcode.com/uid/216506) said in Median of two sorted arrays (https://discuss.leetcode.com/post/238941):

> @lechsa (https://discuss.leetcode.com/uid/389167) Combining two arrays will give O(m+n) time complexity. Anyway, considering your code you are missing one important feature: given arrays are sorted. `(nums1 + nums2).sort()` is a very wrong way of merging two sorted arrays.
>
> `nums1 + nums2` is O(m+n), where `m = len(nums1)` and `n = len(nums2`, and sort is O(n*log(n)).
>
> So, at least to optimize your merging part and you will get O(m+n) total time complexity.

It's already O(m+n). Python uses Timsort, which will recognize the two sorted parts as such and simply merge them. So that's not a "very wrong way" but likely the best way (you're not going to beat Python's Timsort C implementation with your Python code). Edit: "best way" to explicitly merge, I mean. Not the best way to approach the given problem, of course.

L
(https://discuss.leetcode.com/user/lechsa)

lechsa commented 5 days ago

@joysword (https://discuss.leetcode.com/uid/5037) okay, thank you!

@pvlbzn (https://discuss.leetcode.com/uid/216506) thank you, appreciate your input. Let me see if I can optimize the merge, but O(m+n) is less efficient than O(log (m+n)) right? So I think the solution above is the most efficient.

@bhatiagulshan1994 (https://discuss.leetcode.com/uid/303945) will try to find if I can optimize the merge, but the above answer I think is the most optimal.

B
(https://discuss.leetcode.com/user/bhatiagulshan1994)

bhatiagulshan1994 commented 5 days ago

Even I had implemented the similar logic in Java:

1. merge the two arrays
2. sort them
3. find the median

   but the sort method will give O(nlogn) complexity which is not the optimal solution
   Although the solution gets accepted.
   Any idea how the implementation can be made simpler for provided complexity.

P
(https://discuss.leetcode.com/user/pvlbzn)

pvlbzn commented 5 days ago

@lechsa (https://discuss.leetcode.com/uid/389167) Combining two arrays will give O(m+n) time complexity. Anyway, considering your code you are missing one important feature: given arrays are sorted. `(nums1 + nums2).sort()` is a very wrong way of merging two sorted arrays.

`nums1 + nums2` is O(m+n), where `m = len(nums1)` and `n = len(nums2`, and sort is O(n*log(n)).

So, at least to optimize your merging part and you will get O(m+n) total time complexity.

J
(https://discuss.leetcode.com/user/joysword)

joysword commented 6 days ago

@lechsa (https://discuss.leetcode.com/uid/389167) anything that combines the two arrays does not meet the time complexity requirement of O(log(m+n)) and is not optimal.

lechsa commented last week

My answer is accepted, but I don't know if it's optimal. What do you all think? How about the complexity analysis?

(https://discuss.leetcode.com/user/lechsa)

```
class Solution(object):
    def findMedianSortedArrays(self, nums1, nums2):
        """
        :type nums1: List[int]
        :type nums2: List[int]
        :rtype: float
        """
        nums = nums1 + nums2 # join array
        nums.sort() # sort

        if len(nums) % 2 == 0:
            return (nums[(len(nums) // 2) - 1] + nums[len(nums) // 2]) / 2
        else:
            return nums[(len(nums) - 1) // 2]
```

Tal_UzZ commented last week

JS solution,This is not good program,but it can run. Please help me.

(https://discuss.leetcode.com/user/tal_uzz)

```
var findMedianSortedArrays = function(nums1, nums2) {
    let middle = (nums1.length + nums2.length - 1) / 2, length = nums1.length + nums2.length,
n1 = 0, n2 = 0, x = 0, sum = 0;
    let max = nums1[nums1.length-1]>nums2[nums2.length-1]?nums1[nums1.length-1]:nums2[nums2.l
ength-1];
    for(let i=0;i<=middle+(((length+1)*10/2)%10)/10;i++){
        if((nums1[n1]>nums2[n2]&&n2!=nums2.length)||n1===nums1.length){
            x = nums2[n2++];
        }else{
            x = nums1[n1++];
        }
        if(i ===  middle-(((length+1)*10/2)%10)/10){
            sum += x;
        }
    }
    sum += x;
    return parseFloat((sum / 2).toFixed(5));
};
```

View original thread (https://discuss.leetcode.com/topic/101641)

Load more comments...