December 2, 2021 ▪ Arrays / Data Structure

Median of Two Sorted Arrays of different sizes

Problem Statement: Given **two sorted arrays** arr1 and arr2 of size m and n respectively, return the **median** of the two sorted arrays.

Example 1:

```
Input format: arr1 = [1,4,7,10,12], arr2 = [2,3,6,15]

Output format : 6.00000

Explanation:
Merge both arrays. Final sorted array is [1,2,3,4,6,7,10,12,15]. We know that to find the median we find the mid element. Since, the size of the element is odd. By formula, the median will be at [(n+1)/2]th position of the final sorted array. Thus, for this example, the median is at [(9+1)/2]th position which is [5]th = 6.
```

Example 2:

```
Input: arr1 = [1], arr2 = [2]

Output format:
 1.50000

Explanation:

Merge both arrays. Final sorted array is [1,2]. We know that to find the median we find the mid element. Since, the size of the element is even. By formula, the median will be the mean of elements at [n/2]th and  [(n/2)+1]th position of the final sorted array. Thus, for this example, the median is (1+2)/2 = 3/2 = 1.50000.
```

Solution 1: Naive Approach

Intuition :

The point to observe is that the given arrays are sorted. Our task is to merge them into a sorted array. The word "merge" gives us hints to apply the merge step of merge sort.

Approach :

Take two pointers, each pointing to each array. Take an array of size (m+n) to store the final sorted array. If the first pointed element is smaller than the second one, store that value in an array and move the first pointer ahead by one. Else do the same for the second pointer when the case is vice-versa. Then use the formula to get the median position and return the element present at that position.

Dry Run :

Code:

C++ Code | Java Code

```
#include<bits/stdc++.h>
using namespace std;

float median(int nums1[],int nums2[],int m,int n) {
    int finalArray[n+m];
    int i=0,j=0,k=0;
    while(i<m && j<n) {
        if(nums1[i]<nums2[j]) {
            finalArray[k++] = nums1[i++];
        }
        else {
            finalArray[k++] = nums2[j++];
        }
    }
    if(i<m) {
        while(i<m)
            finalArray[k++] = nums1[i++];
    }
    if(j<n) {
        while(j<n)
            finalArray[k++] = nums2[j++];
    }
    n = n+m;
    if(n%2==1)
        return finalArray[((n+1)/2)-1];
    else return ((float)finalArray[(n/2)-1]+(float)finalArray[(n/2)])/2;
}

int main() {
    int nums1[] = {1,4,7,10,12};
    int nums2[] = {2,3,6,15};
    int m = sizeof(nums1)/sizeof(nums1[0]);
    int n = sizeof(nums2)/sizeof(nums2[0]);
    cout<<"The median of two sorted array is "<<fixed<<setprecision(5)
    <<median(nums1,nums2,m,n);
    return 0;
}
```

The Median of two sorted arrays is 6.00000

Time Complexity :

O(m+n)

Reason – We traverse through both the arrays linearly.

Space Complexity :

O(m+n)

Reason – We store the final array whose size is m+n.

Solution 2: Optimised Naive Approach

We can optimize in space complexity.

Approach :

Similar to the naive approach, instead of storing the final merged sorted array, we can keep a counter to keep track of the required position where the median will exist. First, using the median formula, get a position where the median will exist. Then, follow the above approach and instead of storing elements in another array, we will increase the counter value. When the counter value is equal to the median positions, return element.

Time Complexity : O(m+n)

Reason – We are still traversing both the arrays linearly.

Space Complexity: O(1)

Reason – We do not use any extra array.

Solution 3: Efficient solution

Intuition :

We came up with a naive solution because of the hint that two arrays are sorted and we want elements from merged sorted arrays. If we look into the word "sorted arrays", we can think of a binary solution. Hence, we move to an efficient solution using binary search. But how to apply binary search? Let's look into the thought process.

We know that we will get answers only from the final merged sorted arrays. We figured it out with the naive approach discussed above. We will partition both the arrays in such a way that the left half of the partition will contain elements, which will be there when we merge them, till the median element and rest in the other right half. This partitioning of both arrays will be done by binary search.

Approach :

We will do a binary search in one of the arrays which have a minimum size among the two.

Firstly, calculate the median position with (n+1)/2. Point two-pointer, say low and high, equal to 0 and size of the array on which we are applying binary search respectively. Find the partition of the array. Check if the left half has a total number of elements equal to the median position. If not, get the remaining elements from the second array. Now, check if partitioning is valid. This is only when l1<=r2 and l2<=r1. If valid, return max(l1,l2)(when odd number of elements present) else return (max(l1,l2)+min(r1,r2))/2.

If partitioning is not valid, move ranges. When l1>r2, move left and perform the above operations again. When l2>r2, move right and perform the above operations.

Code :

C++ Code | Java Code

```
#include<bits/stdc++.h>
using namespace std;

float median(int num 1[],int num2[],int m,int n) {
    if(m>n)
        return median(nums2,nums1,n,m);//ensuring that binary search happens on minimum size

    int low=0,high=m,medianPos=((m+n)+1)/2;
    while(low<=high) {
        int cut1 = (low+high)>>1;
        int cut2 = medianPos - cut1;

        int l1 = (cut1 == 0)? INT_MIN:nums1[cut1-1];
        int l2 = (cut2 == 0)? INT_MIN:nums2[cut2-1];
        int r1 = (cut1 == m)? INT_MAX:nums1[cut1];
        int r2 = (cut2 == n)? INT_MAX:nums2[cut2];

        if(l1<=r2 && l2<=r1) {
            if((m+n)%2 != 0)
                return max(l1,l2);
            else
                return (max(l1,l2)+min(r1,r2))/2.0;
        }
        else if(l1>r2) high = cut1-1;
        else low = cut1+1;
    }
```

```
    return 0.0;
}

int main() {
    int nums1[] = {1,4,7,10,12};
    int nums2[] = {2,3,6,15};
    int m = sizeof(nums1)/sizeof(nums1[0]);
    int n = sizeof(nums2)/sizeof(nums2[0]);
    cout<<"The Median of two sorted arrays is"<<fixed<<setprecision(5)
    <<median(nums1,nums2,m,n);
    return 0;
}
```

Output:

The Median of two sorted arrays is 6.00000

Time Complexity : O(log(m,n))

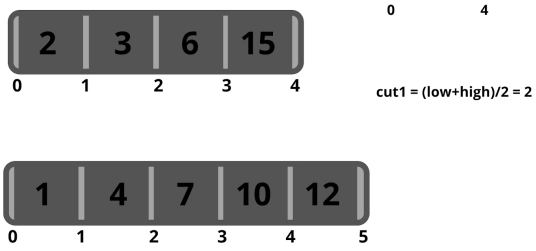
Reason – We are applying binary search on the array which has a minimum size.

Space Complexity: O(1)

Reason – No extra array is used.

Dry Run :

Let's look into the dry run. For example 1, the total size is equal to 9 which is odd. Applying the formula, the median will be at (5+1)/2 = 10/2 = 5th position of the final merged sorted array. The size of nums2 is less than the size of nums1. Thus, apply binary search in nums2.

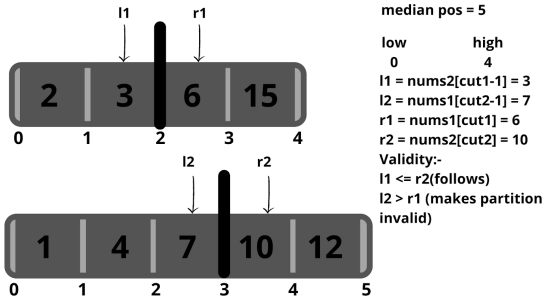


median pos = 5

low high
0 4

cut1 = (low+high)/2 = 2

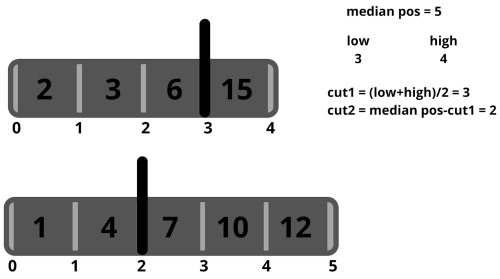
We will cut1 to be at pos 2. So, the total number of elements at the left of cut1 is 2. So, we will choose the remaining 3 elements from num2. We will make cut2 at 3.



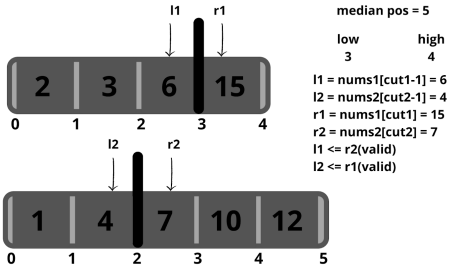
median pos = 5

low high
0 4
l1 = nums2[cut1-1] = 3
l2 = nums1[cut2-1] = 7
r1 = nums1[cut1] = 6
r2 = nums2[cut2] = 10
Validity:-
l1 <= r2(follows)
l2 > r1 (makes partition invalid)

Partitioning is invalid. l2>r1 and to make the left half valid, we have to decrease the value of l2. We have to move right by moving low to cut1+1.



We moved towards the right and got a new partitioning.



We can see it is a valid left half. Thus, we get our median as max(l1,l2).

> Special thanks to **Dewanshi Paul** for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, please check out this article

sorting

« Previous Post
Check if two trees are identical

Next Post »
Serialize And Deserialize a Binary Tree

Load Comments