medium.com

Finding the Median of 2 Sorted Arrays in Logarithmic Time

hamid

12-15 minutes

This problem is featured on LeetCode along with a fairly clever solution that is explained in a somewhat intricate way. This post is an attempt to explain the general intuition behind that solution in simple terms.

Main Challenge

The crux of this problem is finding what two arrays would look like when they are merged, without actually merging them since this would take *O(n+m)* time.

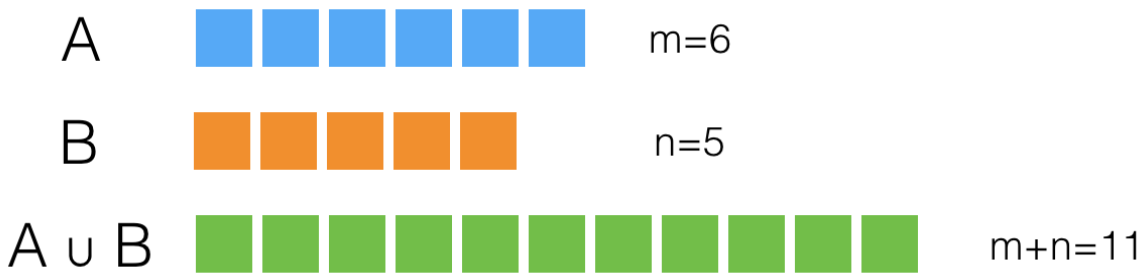


Fig. 1 — Two sorted arrays, A and B, whose lengths are m and n, respectively. A ∪ B is a third array that represents the result of merging A and B.

Practically speaking, we are only interested in knowing what the

left half of the merged array, A ∪ B, would be, because this is the subarray that ends with the median.

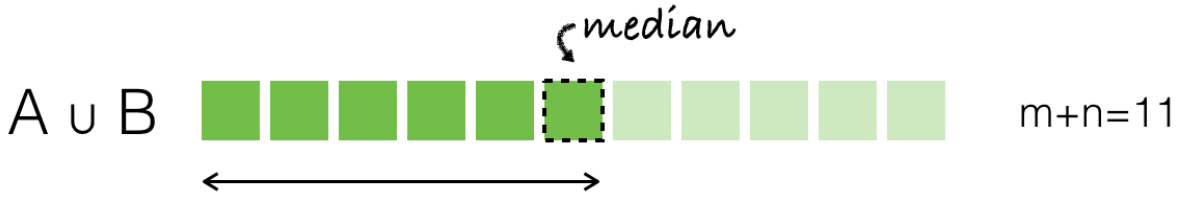


Fig. 2 — We are only interested in the left half (size = 6) of the merged array.

Key Questions

Question 1

Is there a way we can guess what the values in the left half of A ∪ B would be without merging A and B?

Let's think about it. What do we know about this half? We know:

- It will contain six values.
- These six values could be coming from A, B, or both.

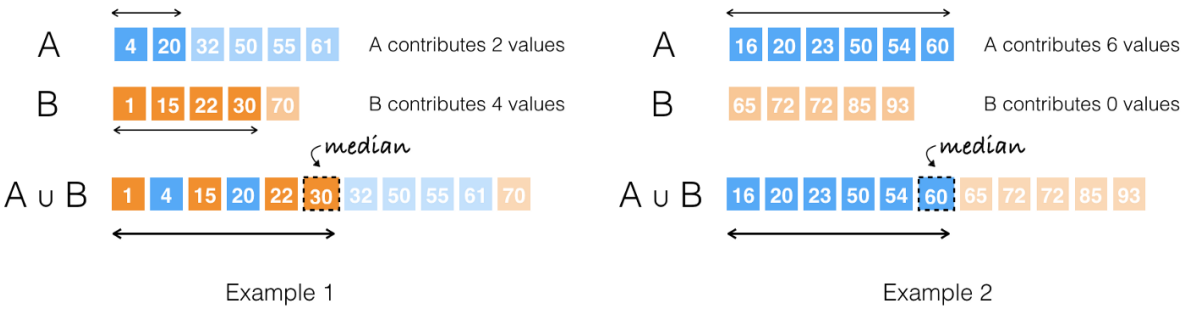


Fig. 3 — A couple of examples illustrating how the values within A and B determine the number of elements each array contributes to the left half of A ∪ B.

Not knowing anything about their values, A and B could contribute to the left half of A ∪ B in six different ways.

Fig. 4— An exhaustive list of the different ways A and B could contribute to the left half of their merged result.

Question 2

How is knowing the number of values contributed by A and B to the left half of A ∪ B going to help us find the median?

Well,

- We could just compare the last value contributed by A with the last value contributed by B. The greater of the two would be the median.
- In cases where either array contributes zero elements — like the one in the lower right corner of Fig. 4 — the median will be the last value contributed by the other array.

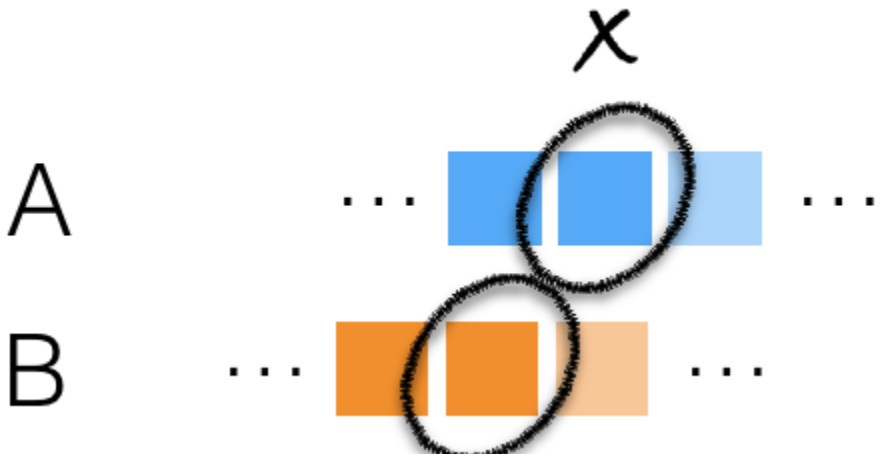


Fig. 6 — The greater of the 2 values contributed by A and B, labeled *x* and *y*, will be the median if A and B are merged.

Question 3

What if *m* + *n* is an even number?

We would still need to know the last value in the left half of A ∪ B. The only difference is that we will need to know just one more value thereafter in order to compute the final value of the median.

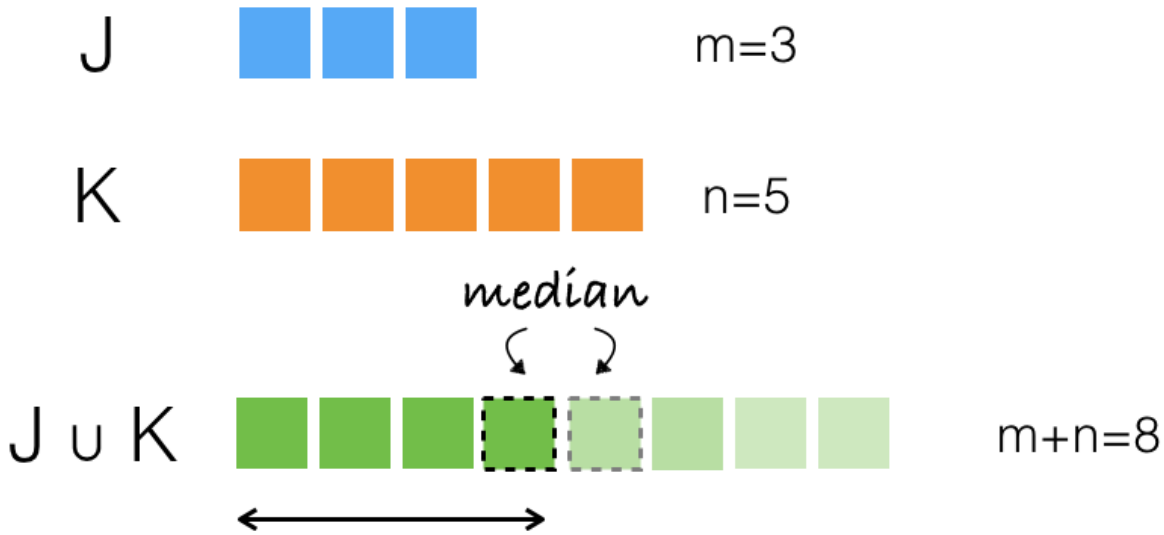


Fig. 7 — Another example with two arrays, J and K, whose combined size is even. We still need to identify the last value in the left half of their merged result. We'll just need to find out the value next to it.

Question 4

If there are six different ways A and B can contribute values to the

left half of A ∪ B, how do we know the correct one?

To answer this question, let's look at a few examples.

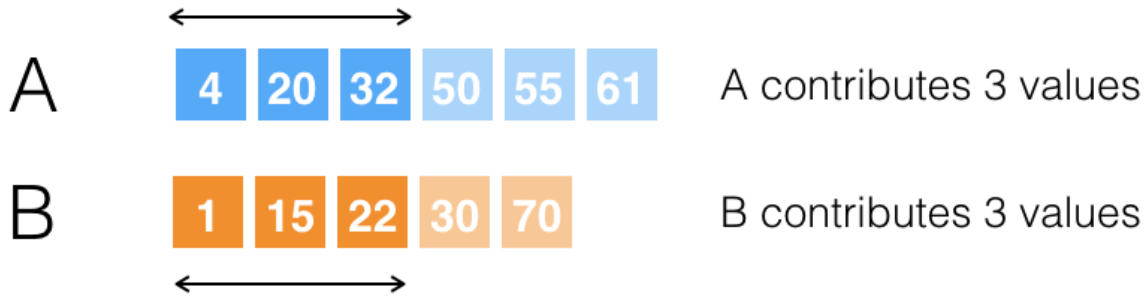
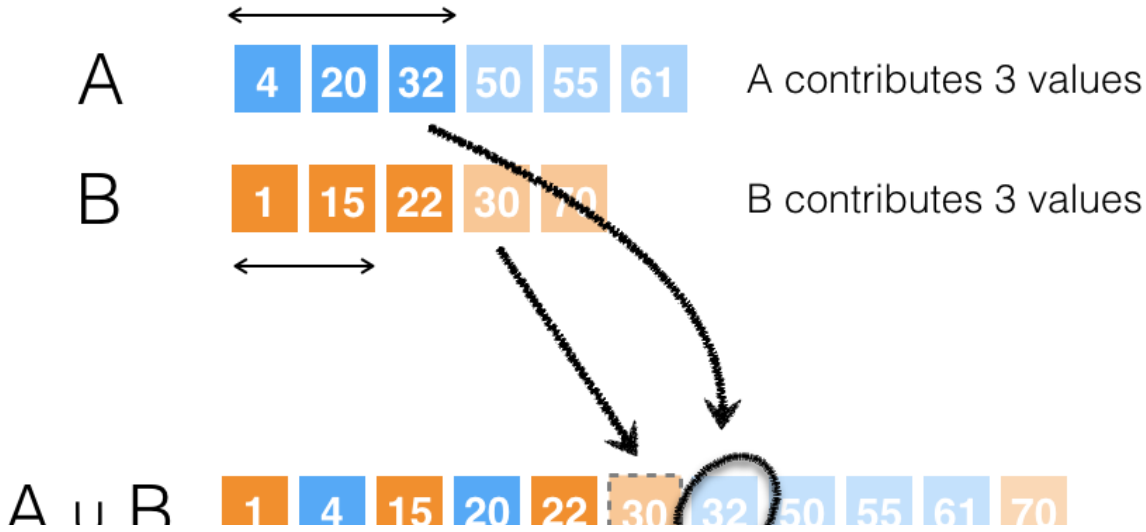


Fig. 8 — An example where we examine the possibility of arrays A and B contributing 3 values each, to the left half of A ∪ B.

We know based on the lengths of A and B that the left half of A ∪ B is of length six. We also know that the median will be the sixth value in A ∪ B. Let's assume A and B will contribute three elements each, to the left half of A ∪ B, as shown in Fig. 8.

To verify this assumption, we examine the greatest value contributed by each of A and B, i.e. 32 and 22. Since 32 is greater, we expect it to show after 22 in A ∪ B. So, is it safe to say 32 will be the sixth value and the median? Unfortunately not. The reason is that when A and B are merged, 32 will not appear until after the value 30 from array B, since the merged array has to be sorted.



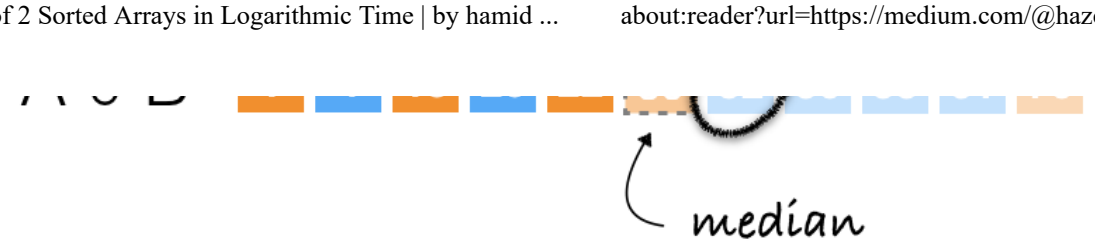


Fig. 9 — The value 32 will not appear in A ∪ B until after the value 30 from array B.

It turns out it wasn't sufficient to compare 32 and 22. We should have also compared 32 to the value next to 22, i.e. 30, to make sure 32 won't be pushed any further in A ∪ B. Generally speaking, it is not enough to compare the greatest values contributed by A and B, but we also need to make sure the greater of the two won't be pushed further away by some value in the other array.

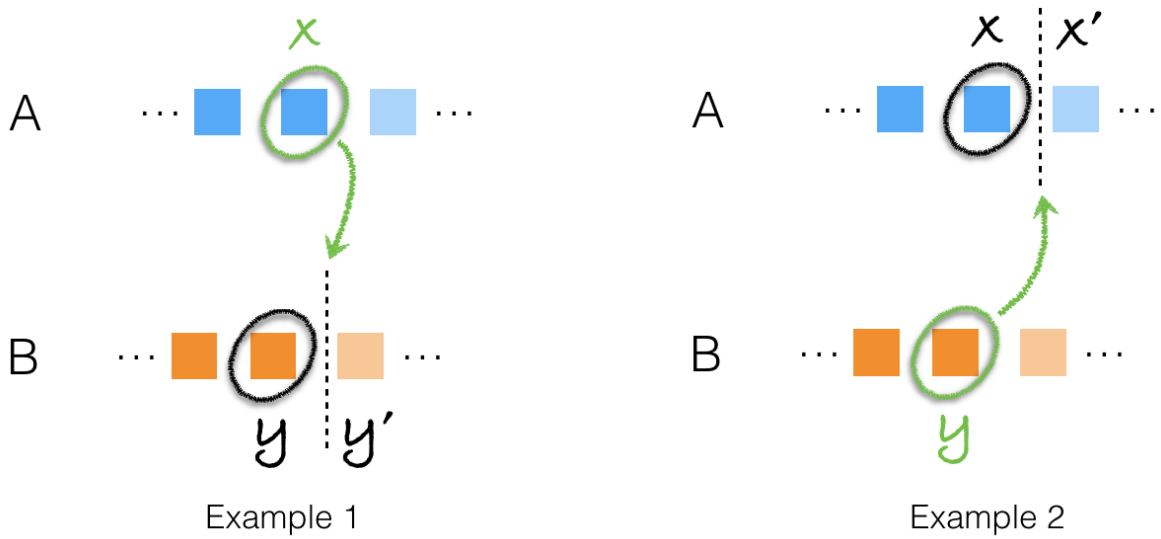


Fig. 10 — Example 1 illustrates the case where the greatest value contributed by A, labelled x, is greater than the greatest value contributed by B, labelled y. To confirm x is the median, we need to compare it with the value next to y, labelled y', so we are certain x lies between y and y', and will not be pushed further along in A ∪ B. Example 2 illustrates the case where y is greater than x.

Going back to the example in Fig. 9, 32 is greater than 22 but it is also greater than 22's successor, 30. This implies that 32 is not the median.

Knowing that B has a smaller value to offer than A's greatest contributed value, 32, strongly suggests B will contribute more values to the left half of A ∪ B. Therefore, we should consider increasing the number of values contributed by B, thus decreasing the number of values contributed by A. As show in Fig. 11, allowing B to contribute four values instead of three causes A's contribution to shrink and, more importantly, reveals that 30 is the median.

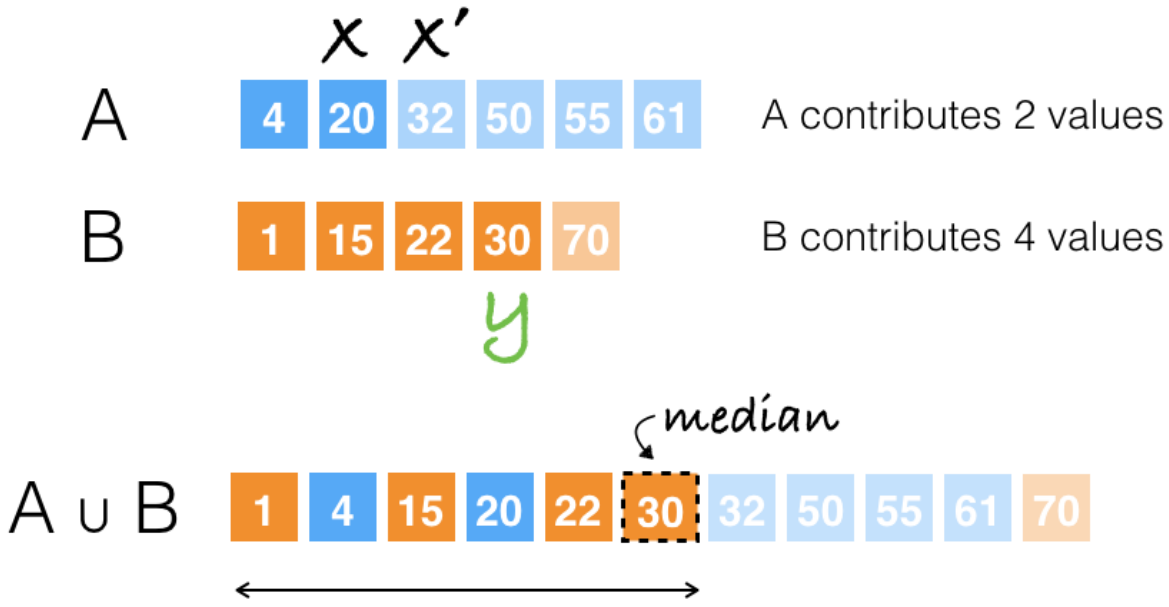


Fig. 11 — When A and B contribute 2 and 4 values, respectively, the greatest value contributed by B, y = 30, is greater than the greatest value contributed by A, x = 20. y is also smaller than x' = 32, which indicates y is indeed the median.

Question 5

Now that we have a way to identify the correct split between A and B, can't we just try all the possible splits to find the median?

We can, but examining every possible split means the amount of work we will be doing is still linearly proportional to m + n. For

instance, in the example we have been studying, there were six different ways A (size = 6) and B (size = 5) could contribute to the left half of A ∪ B. We would like our algorithm to have logarithmic runtime complexity.

Question 6

How do we find the correct split in logarithmic time?

We use the concept of binary search to reduce the number of possibilities we consider. It may not be very obvious how binary search is relevant, so let's take a moment to understand this.

In essence, what we're trying to find is the number of values each of A and B will contribute to the left half of A ∪ B. But since we know the size of this half in advance, $(m + n)/2$, we can simplify our objective by saying we're only interested in the number of values A is contributing. For instance, in our example, if we know A is contributing four values, then it follows that B is contributing two, since the left half of A ∪ B has a total length of six.

This leads us to the following question: what is the minimum and maximum number of values can A contribute? In our example, A must contribute at least one value; the size of the left half of A ∪ B is six, and B has five values only. On the other hand, A can contribute all of its six values to the left half of A ∪ B, which could happen if all the values in A were smaller than those in B. This is to say we can find the median of A ∪ B if we know A's contribution size, which is an integer in the range [1, 6]. Now instead of trying out all the possible sizes from 1 to 6, we can use binary search, i.e.

1. Set A's minimum and maximum contribution sizes to 1 and 6,

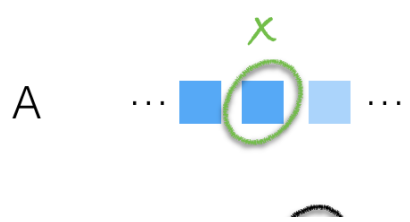
respectively (*min = 1, max = 6*).

2. Consider the midpoint between min and max, *mid* = (1 + 6)/2 = 3. Check to see if our conditions for finding the median are met if A's contribution size is equal to *mid* (by performing the comparisons we discussed in the answer to question 4). If so, then we found the solution, and we know the median is the greater of the greatest values contributed by A and B.
3. Otherwise, we can adjust min to *mid + 1* or *max* to *mid − 1* based on comparing A's greatest contributed value, B's greatest contributed value, and the value that succeeds the smaller of the two.

Question 7

How do we know whether to increase or decrease A's contribution size?

- If $y < x \leq y'$ then we found the solution, and x is the median.
- If $x < y \leq x'$ then we found the solution, and y is the median.
- If $x > y$ and $x > y'$ then we need to decrease A's contribution size because x will end up beyond the left half of A ∪ B. It's useful to observe that if $x > y'$ then $x > y$ must be true since $y' > y$.
- If $y > x$ and $y > x'$ then we need to decrease B's contribution size, i.e. increase A's contribution size, because y will end up beyond the left half of A ∪ B. It's also useful to observe that checking if $y > x'$ should be sufficient.



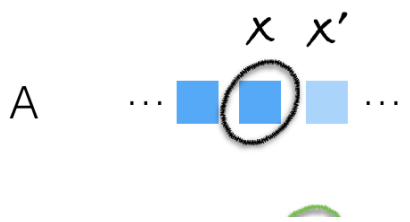


Fig. 12 — x is the median if it lies between y and y'. y is the median if it lies between x and x'. Otherwise, contributions of A/B need to be readjusted.

General Approach

Listing 1 below shows an initial implementation of the solution we have discussed so far, written in C#. Certain operations have been deliberately abstracted behind functions whose implementations are not included. These operations are discussed in detail in the following sections. But before we move on, please take a moment to look at the code below and understand it thoroughly.

Listing 1 — An initial implementation

The similarities in the overall structure between this algorithm and binary search should be clear by now. This implementation has a number of issues that we need to address nonetheless.

Input Validation

The code in Listing 1 does not check if the input arrays are null or empty. Checking for null is trivial, and so is checking if both arrays are empty. If only one of them is empty, we can directly compute the median of the other.

Computing Left Half Length

In the example we have been studying so far, we chose a left half

of length 6 for two input arrays whose combined size is 11. The main advantage of this decision is that the median becomes the last element in this half. We also discussed how we can generalize this to work in case of even lengths.

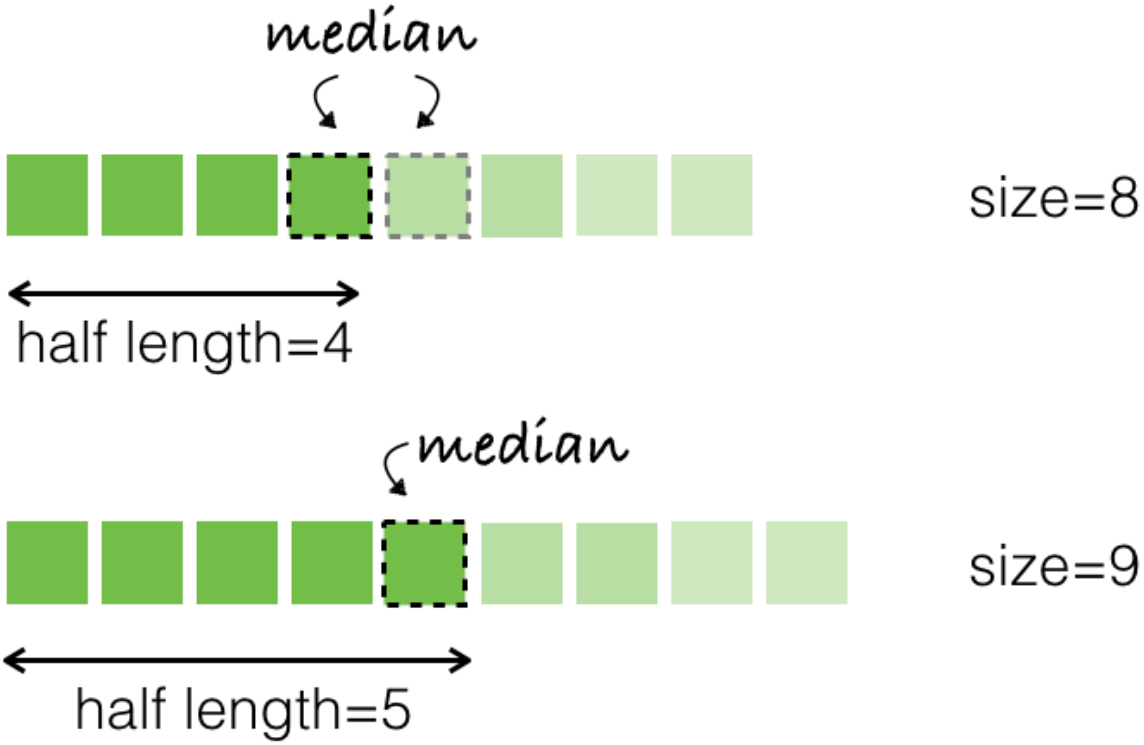


Fig. 13– Two example arrays of even and odd lengths. Left half length is 4 for the first array and 5 for the second.

To tackle both cases, we can compute the left half length using either of the two equations below.

$$Left\ half\ length\ =\ \left\lceil \frac{m+n}{2} \right\rceil \quad (floating\ point\ division)$$

$$Left\ half\ length\ =\ \frac{m+n+1}{2} \quad (integer\ division)$$

Fig. 14 — Equations for computing the left half length of the merged array, A ∪ B.

Min and Max Number of Values to Contribute

Deciding the initial values of *aMinCount* and *aMaxCount* is both important and subtle. In the example of arrays A and B, we observed that *aMinCount* and *aMaxCount* were equal to 1 and 6, respectively. A couple of things worth highlighting in this example are:

- Zero isn't a valid value for *aMinCount* since B has five values only. There is no way B can contribute enough values to fill all six slots of the left half of A ∪ B on its own.
- It's very useful to observe we can conduct the search in B instead of A, thus defining *bMinCount* and *bMaxCount* rather than *aMinCount* and *aMaxCount.* However, it's also important to realize *bMinCount* and *bMaxCount* won't have the same initial range, [1, 6], as *aMinCount* and *aMaxCount*. This follows from the fact that, unlike A, B may contribute no values to the left half of A ∪ B (i.e. if all values in A are smaller than those in B), or all of its five values (i.e. if all values in B are smaller than those in A). In addition to having the simpler range of [0, m=5] for *bMinCount* and *bMaxCount*, searching B has the added benefit of having fewer values to examine.

Computing Midpoint

We compute *aCount* as (*aMinCount* + *aMaxCount*)/2, which is susceptible to overflows if the values of *aMinCount* and *aMaxCount* are close to the maximum allowable integer value, e.g. when we're searching the far right extents of a very large array. This is a general issue that all algorithm implementations based on binary search are susceptible to. You can read more about it in this article by Joshua Bloch.

Guarding Against Invalid Index Errors

An important detail we need to take into account as we compare x, y, x', and y', is that some of them may be non-existent/undefined. For instance, if A is contributing all of its six values to the left half of A ∪ B, then

- y will be undefined since B is not contributing any values, and
- x' will also be undefined because there are no elements left in A.

In this case, expressions like *B[bCount − 1]* and *A[aCount]* will yield index out of bounds/range errors, since *bCount* and *aCount* will have the values 0 and *A.Length*, respectively.

We certainly need some additional checks to account for such cases. Alternatively, the code in Listing 2 addresses this issue by utilizing a nifty C# feature — nullable types. If you are not familiar with C#, it's sufficient to know that:

1. A nullable integer, as its name suggests, can be null.
2. Any less-than/greater-than comparisons involving a null nullable, directly evaluate to false.

Listing 2 — The final implementation

Of course you can replace these nullables with simple index checks if you're solving this problem using a different language. An example is provided in Listing 3 below.

Listing 3 — An implementation that does not utilize C#'s nullable types.

I hope this post helped you establish a better understanding of a fairly interesting and intricate problem. One last thing to note is that I do not think a problem like this is appropriate for typical

1-hour interview sessions. Solving such a problem requires a great deal of reflection and an even greater deal of validation. However, this is a post for another day.

Thanks for reading.