Brute Force Approach
Algorithm / Intuition

Naive Approach(Brute-force):

The extremely naive approach is to merge the two sorted arrays and then find the median in that merged array.

How to merge two sorted arrays:

The word “merge” suggests applying the merge step of the merge sort algorithm. In that step, we essentially perform the same actions as required by this solution. By using two pointers on two given arrays, we fill up the elements into a third array.

How to find the median:

- **If the length of the merged array (n1+n2) is even:** The median is the average of the two middle elements. index = (n1+n2) / 2, median = (arr3[index] + arr3[index-1]) / 2.0.
- **If the length of the merged array (n1+n2) is odd:** index = (n1+n2) / 2, median = arr3[index].

Algorithm:

1. We will use a third array i.e. arr3[] of size (n1+n2) to store the elements of the two sorted arrays.
2. Now, we will take two pointers i and j, where i points to the first element of arr1[] and j points to the first element of arr2[].
3. Next, using a while loop(while(i < n1 && j < n2)), we will select two elements i.e. arr1[i] and arr2[j], and consider the smallest one among the two. Then, we will insert the smallest element in the third array and increase that specific pointer by 1.
    1. **If arr1[i] < arr2[j]:** Insert arr1[i] into the third array and increase i by 1.
    2. **Otherwise:** Insert arr2[j] into the third array and increase j by 1.
4. After that, the left-out elements from both arrays will be copied as it is into the third array.
5. Now, the third array i.e. arr3[] will be the sorted merged array. Now the median will be the following:
    1. **If the length of arr3[] i.e. (n1+n2) is even:** The median is the average of the two middle elements. index = (n1+n2) / 2, median = (arr3[index] + arr3[index-1]) / 2.0.
    2. **If the length of arr3[] i.e. (n1+n2) is odd:** index = (n1+n2) / 2, median = arr3[index].
6. Finally, we will return the value of the median.

Dry-run: *Please refer to the attached video for a detailed dry-run.*

Code

C++JavaPythonJavaScript

```
import java.util.*;

public class tUf {
    public static double median(int[] a, int[] b) {
        // Size of two given arrays
        int n1 = a.length;
        int n2 = b.length;

        List<Integer> arr3 = new ArrayList<>();
        // Apply the merge step
        int i = 0, j = 0, k = 0;
        while (i < n1 && j < n2) {
            if (a[i] < b[j]) {
                arr3.add(a[i++]);
            } else {
                arr3.add(b[j++]);
            }
        }

        // Copy the left-out elements
        while (i < n1) {
            arr3.add(a[i++]);
        }
        while (j < n2) {
            arr3.add(b[j++]);
        }

        // Find the median
        int n = n1 + n2;
        if (n % 2 == 1) {
            return (double) arr3.get(n / 2);
        }
```

```
        double median = ((double) arr3.get(n / 2) + (double) arr3.get((n / 2) -
1)) / 2.0;
        return median;
    }

    public static void main(String[] args) {
        int[] a = {1, 4, 7, 10, 12};
        int[] b = {2, 3, 6, 15};
        System.out.println("The median of two sorted arrays is " + median(a,
b));
    }
}
```

Output: The median of two sorted arrays is 6.0

Complexity Analysis

Time Complexity: O(n1+n2), where n1 and n2 are the sizes of the given arrays.
Reason: We traverse through both arrays linearly.

Space Complexity: O(n1+n2), where n1 and n2 are the sizes of the given arrays.
Reason: We are using an extra array of size (n1+n2) to solve this problem.

Better Approach
Algorithm / Intuition

To optimize the space used in the previous approach, we can eliminate the third array used to store the final merged result. After closer examination, we realize that we only need the two middle elements at indices (n1+n2)/2 and ((n1+n2)/2)-1, rather than the entire merged array, to solve the problem effectively.

We will stick to the same basic approach, but instead of storing elements in a separate array, we will use a counter called 'cnt' to represent the imaginary third array's index. As we traverse through the arrays, when 'cnt' reaches either index (n1+n2)/2 or ((n1+n2)/2)-1, we will store that particular element. This way, we can achieve the same goal without using any extra space.

Algorithm:

1. We will call the required indices as **ind2 =** (n1+n2)/2 and **ind1 =** ((n1+n2)/2)-1. Now we will declare the counter called ‘cnt’ and initialize it with 0.
2. Now, as usual, we will take two pointers i and j, where i points to the first element of arr1[] and j points to the first element of arr2[].
3. Next, using a while loop(while(i < n1 && j < n2)), we will select two elements i.e. arr1[i] and arr2[j], and consider the smallest one among the two. Then, we will increase that specific pointer by 1.
   In addition to that, in each iteration, we will check if the counter ‘cnt’ hits the indices **ind1 or ind2.** when 'cnt' reaches either index ind1 or ind2, we will store that particular element. We will also increase the ‘cnt’ by 1 every time regardless of matching the conditions.
   1. **If arr1[i] < arr2[j]:** Check ‘cnt’ to perform necessary operations and increase i and ‘cnt’ by 1.
   2. **Otherwise:** Check ‘cnt’ to perform necessary operations and increase j and ‘cnt’ by 1.

4. After that, the left-out elements from both arrays will be copied as it is into the third array. While copying we will again check the above-said conditions for the counter, ‘cnt’ and increase it by 1.
5. Now, let’s call the elements at the required indices as **ind1el(*at ind1*)** and **ind2el(*at ind2*)**:
    1. **If the total length i.e. (n1+n2) is even:** The median is the average of the two middle elements. median = (ind1el + ind2el) / 2.0.
    2. **If the total length i.e. (n1+n2) is odd:** median = ind2el.
6. Finally, we will return the value of the median.

Dry-run: *Please refer to the attached video for a detailed dry-run.*

Code
C++JavaPythonJavaScript

```
import java.util.*;

public class tUf {
    public static double median(int[] a, int[] b) {
        // Size of two given arrays
        int n1 = a.length;
        int n2 = b.length;

        int n = n1 + n2; //total size
        //required indices:
        int ind2 = n / 2;
        int ind1 = ind2 - 1;
        int cnt = 0;
        int ind1el = -1, ind2el = -1;

        //apply the merge step:
        int i = 0, j = 0;
        while (i < n1 && j < n2) {
            if (a[i] < b[j]) {
                if (cnt == ind1) ind1el = a[i];
                if (cnt == ind2) ind2el = a[i];
                cnt++;
                i++;
            } else {
                if (cnt == ind1) ind1el = b[j];
                if (cnt == ind2) ind2el = b[j];
                cnt++;
                j++;
            }
        }

        //copy the left-out elements:
        while (i < n1) {
            if (cnt == ind1) ind1el = a[i];
            if (cnt == ind2) ind2el = a[i];
            cnt++;
            i++;
        }
        while (j < n2) {
            if (cnt == ind1) ind1el = b[j];
            if (cnt == ind2) ind2el = b[j];
            cnt++;
            j++;
```

```
        }

        //Find the median:
        if (n % 2 == 1) {
            return (double)ind2el;
        }

        return (double)((double)(ind1el + ind2el)) / 2.0;
    }

    public static void main(String[] args) {
        int[] a = {1, 4, 7, 10, 12};
        int[] b = {2, 3, 6, 15};
        System.out.println("The median of two sorted arrays is " + median(a,
b));
    }
}
```

Output: The median of two sorted arrays is 6.0

Complexity Analysis

Time Complexity: O(n1+n2), where n1 and n2 are the sizes of the given arrays.
Reason: We traverse through both arrays linearly.

Space Complexity: O(1), as we are not using any extra space to solve this problem.

Optimal Approach
Algorithm / Intuition

We are going to use the Binary Search algorithm to optimize the approach.

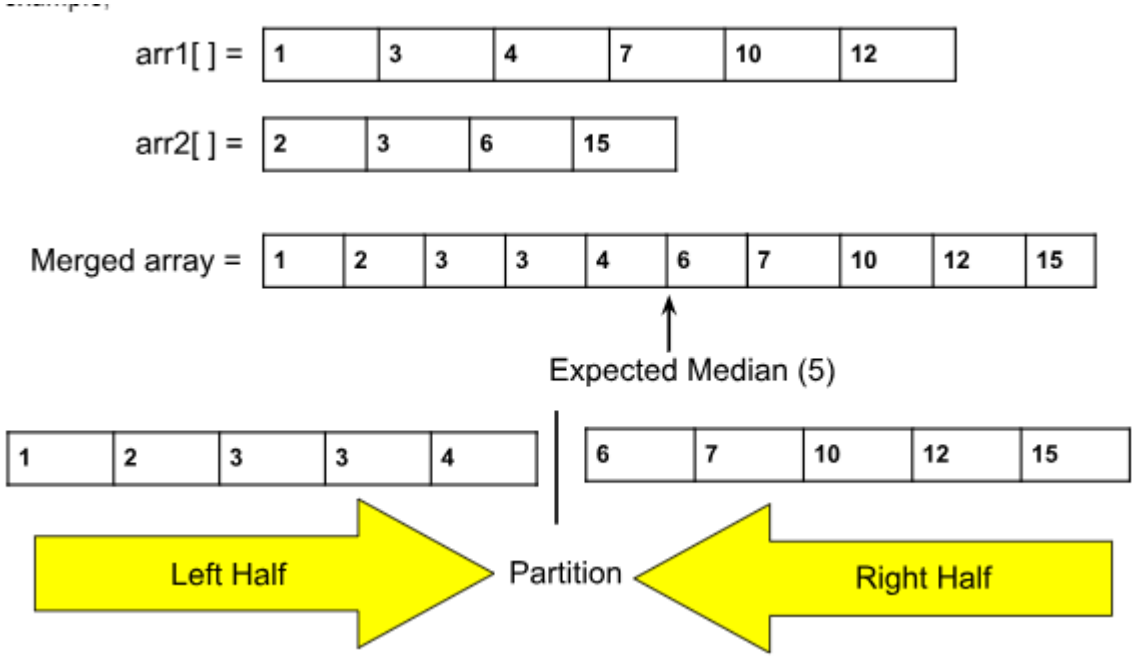
The primary objective of the Binary Search algorithm is to efficiently determine the appropriate half to eliminate, thereby reducing the search space by half. It does this by determining a specific condition that ensures that the target is not present in that half.

Now, let's learn through the following observations how we can apply binary search to this problem. First, we will try to solve this problem where n1+n2 is even and then we will consider the odd scenario.

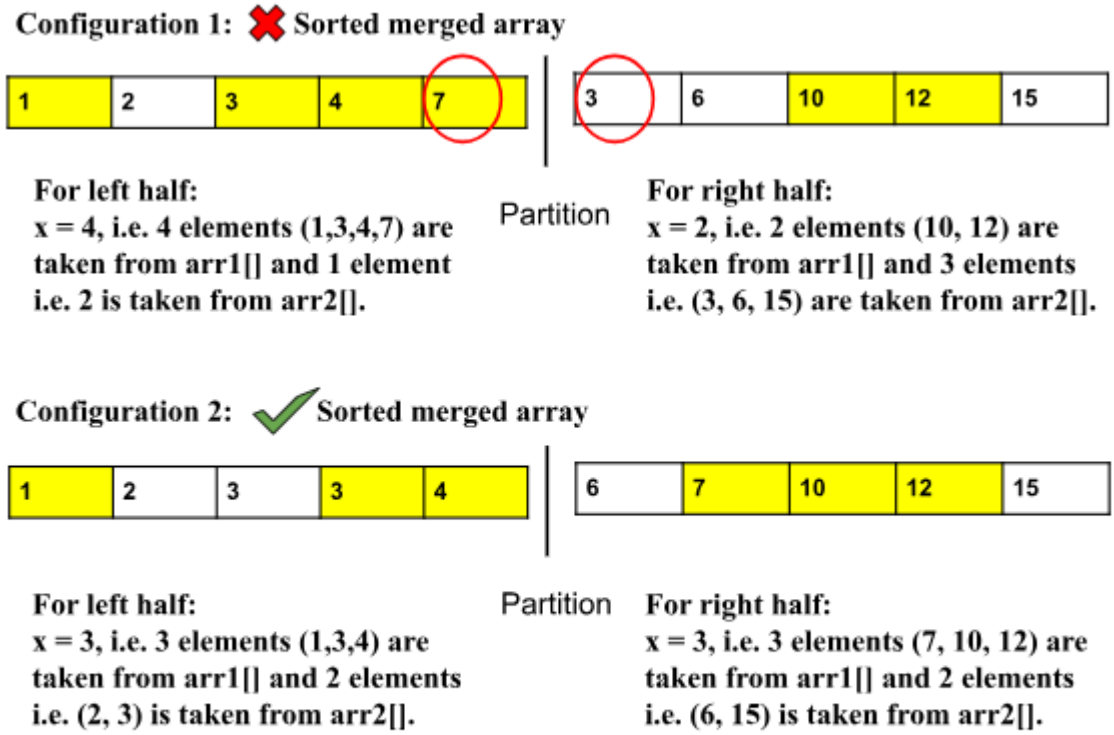
Observations:

Assume, n = n1+n2 i.e. the total length of the final merged array.

- **Median creates a partition on the final merged array:** Upon closer observation, we can easily show that the median divides the final merged array into two halves. For example,



- **Characteristics of each half:**
  - Each half contains (n/2) elements.
  - Each half also contains **x** elements from the first array i.e. arr1[] and **(n/2)-x** elements from the second array i.e. arr2[]. The value of x might be different for the two halves. For example, in the above array, the left half contains 3 elements from arr1[] and 2 elements from arr2[].
- **The unique configuration of halves:** Considering different values of x, we can get different left and right halves(x = the number of elements taken from arr1[] for a particular half). Some different configurations for the above example are shown below:

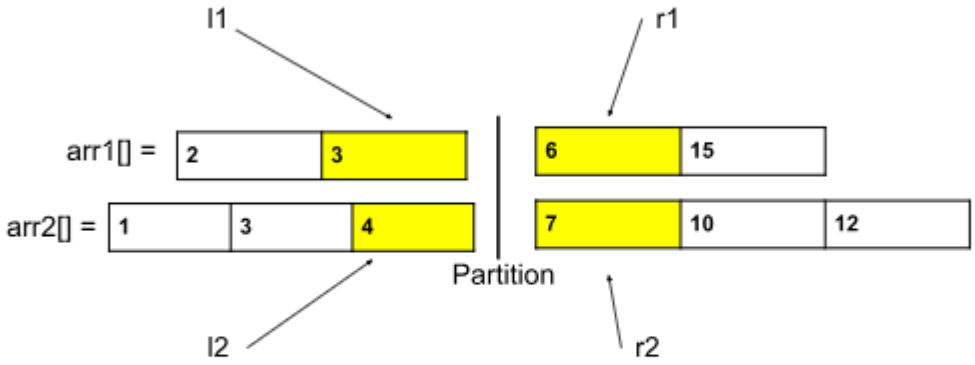


The first configuration is not valid as the merged array containing the left and right half is not sorted. But the second one is valid. So, for a valid merged array, the configuration of the left and right halves is unique.

How to solve the problem using the above observations:

- **Try to form the unique left half:**
  - For a valid merged array, the configurations of the two halves are unique. So, we can try to form the halves with different values of **x,** where x = the number of elements taken from arr1[] for a particular half.
  - There's no need to construct both halves. Once we have the correct left half, the right half is automatically determined, consisting of the remaining elements not yet considered. ***Therefore, our focus will solely be on creating the unique left half.***
  - **How to form all configurations of the left half:** We know that the left half will surely contain **x** elements from arr1[] and **(n/2)-x** elements from arr2[]. Here the only variable is **x.** The minimum possible value of x is 0 and the maximum possible value is n1(*i.e. The length of the considered array*).
    For all the values,[0, n1] of x, we will try to form the left half and then we will check if that half's configuration is valid.
- **Check if the formed left half is valid:** For a valid left half, the merged array will always be sorted. So, if the merged array containing the formed left half is sorted, the formation is valid.
  - **How to check if the merged array is sorted without forming the array:**
    In order to check we will consider 4 elements, i.e. l1, l2, r1, r2.
    - l1 = the maximum element belonging to arr1[] of the left half.
    - l2 = the maximum element belonging to arr2[] of the left half.
    - r1 = the minimum element belonging to arr1[] of the right half.
    - r1 = the minimum element belonging to arr2[] of the right half.

For example,



If max(l1, l2) <= min(r1, r2), then the merged array will always be sorted.

If the value of x is 2 and n/2 is 5, l1 = arr1[1], l2 = arr2[2], r1 = arr1[2], r2 = arr2[3]. Thus we can check if the formed left half is valid or not.

How to apply Binary search to form the left half:

- We will check the formation of the left half for all possible values of **x.** Now, we know that the minimum possible value of x is 0 and the maximum is n1(*i.e. The length of the considered array*). Now the range is sorted. So, **we will apply the binary search on the possible values of x i.e. [0, n1].**
- **How to eliminate the halves based on the values of x:** Binary search works by eliminating the halves in each step. Upon closer observation, we can eliminate the halves based on the following conditions:
    - **If l1 > r2:** This implies that we have considered more elements from arr1[] than necessary. So, we have to take less elements from arr1[] and more from arr2[]. In such a scenario, we should try smaller values of x. To achieve this, we will eliminate the right half (high = mid-1).
    - **If l2 > r1:** This implies that we have considered more elements from arr2[] than necessary. So, we have to take less elements from arr2[] and more from arr1[]. In such a scenario, we should try bigger values of x. To achieve this, we will eliminate the left half (low = mid+1).

Until now, we have learned how to use binary search but with the assumption that (n1+n2) is even. Let's generalize this.

If (n1+n2) is odd: In the case of even, we have considered the length of the left half as (n1+n2) / 2. In this case, that length will be (n1 + n2 + 1) / 2. This much change is enough to handle the case of odd. The rest of the things will be completely the same.
As in the code, division refers to integer division, this modified formula (n1+n2+1) / 2 will be valid for both cases of odd and even.

What will be the answer i.e. the median:

- **If l1 <= r2 && l2 <= r1:** This condition assures that we have found the correct elements.
    - **If (n1+n2) is odd:** The median will be max(l1, l2).
    - **Otherwise,** median = (max(l1, l2) + min(r1, r2)) / 2.0

Note: We are applying binary search on the possible values of x i.e. [0, n1]. Here n1 is the length of arr1[]. Now, to further optimize it, we will consider the smaller array as arr1[]. So, the actual range will be [0, min(n1, n2)].

Algorithm:

1. First, we have to make sure that the arr1[] is the smaller array. If not by default, we will just swap the arrays. Our main goal is to consider the smaller array as arr1[].
2. **Calculate the length of the left half:** left = (n1+n2+1) / 2.
3. **Place the 2 pointers i.e. low and high:** Initially, we will place the pointers. The pointer low will point to 0 and the high will point to n1(*i.e. The size of arr1[]*).
4. **Calculate the 'mid1' i.e. x and 'mid2' i.e. left-x:** Now, inside the loop, we will calculate the value of 'mid1' using the following formula:
   mid1 = (low+high) // 2 ('//' refers to integer division)
   mid2 = left-mid1
5. **Calculate l1, l2, r1, and r2:** Generally,
    1. l1 = arr1[mid1-1]
    2. l2 = arr2[mid2-1]
    3. r1 = arr1[mid1]
    4. r2 = arr2[mid2]

    The possible values of 'mid1' and 'mid2' might be 0 and n1 and n2 respectively. So, to handle these cases, we need to store some default values for these four variables. The default value for l1 and l2 will be **INT_MIN** and for r1 and r2, it will be **INT_MAX.**
6. **Eliminate the halves based on the following conditions:**
    1. **If l1 <= r2 && l2 <= r1:** We have found the answer.
        1. **If (n1+n2) is odd:** Return the median = max(l1, l2).
        2. **Otherwise:** Return median = (max(l1, l2)+min(r1, r2)) / 2.0
    2. **If l1 > r2:** This implies that we have considered more elements from arr1[] than necessary. So, we have to take less elements from arr1[] and more from arr2[]. In such a scenario, we should try smaller values of x. To achieve this, we will eliminate the right half (high = mid1-1).
    3. **If l2 > r1:** This implies that we have considered more elements from arr2[] than necessary. So, we have to take less elements from arr2[] and more from arr1[]. In such a scenario, we should try bigger values of x. To achieve this, we will eliminate the left half (low = mid1+1).
7. Finally, outside the loop, we will include a dummy return statement just to avoid warnings or errors.

The steps from 4-6 will be inside a loop and the loop will continue until low crosses high.

Dry-run: *Please refer to the attached video for the dry run.*

Code
C++JavaPythonJavaScript

```
import java.util.*;

public class tUf {
    public static double median(int[] a, int[] b) {
        int n1 = a.length, n2 = b.length;
        //if n1 is bigger swap the arrays:
        if (n1 > n2) return median(b, a);

        int n = n1 + n2; //total length
        int left = (n1 + n2 + 1) / 2; //length of left half
        //apply binary search:
        int low = 0, high = n1;
        while (low <= high) {
            int mid1 = (low + high) / 2;
            int mid2 = left - mid1;
            //calculate l1, l2, r1 and r2;
            int l1 = (mid1 > 0) ? a[mid1 - 1] : Integer.MIN_VALUE;
            int l2 = (mid2 > 0) ? b[mid2 - 1] : Integer.MIN_VALUE;
            int r1 = (mid1 < n1) ? a[mid1] : Integer.MAX_VALUE;
            int r2 = (mid2 < n2) ? b[mid2] : Integer.MAX_VALUE;

            if (l1 <= r2 && l2 <= r1) {
                if (n % 2 == 1) return Math.max(l1, l2);
                else return ((double) (Math.max(l1, l2) + Math.min(r1, r2))) /
2.0;
            } else if (l1 > r2) high = mid1 - 1;
            else low = mid1 + 1;
        }
        return 0; //dummy statement
    }

    public static void main(String[] args) {
        int[] a = {1, 4, 7, 10, 12};
        int[] b = {2, 3, 6, 15};
        System.out.println("The median of two sorted arrays is " + median(a,
b));
    }
}
```

Output: The median of two sorted array is 6.0

Complexity Analysis

Time Complexity: O(log(min(n1,n2))), where n1 and n2 are the sizes of two given arrays.
Reason: We are applying binary search on the range [0, min(n1, n2)].

Space Complexity: O(1) as no extra space is used.