K-th Element of two sorted arrays

Problem Statement: Given **two sorted arrays** of size **m** and **n** respectively, you are tasked with finding the element that would be at the **kth position** of the **final sorted array**.

Examples :

```
Input: m = 5

       n = 4

       array1 = [2,3,6,7,9]

       array2 = [1,4,8,10]

       k = 5



Output:

 6



Explanation: Merging both arrays and sorted. Final array will be -

 [1,2,3,4,6,7,8,9,10]

We can see at k = 5 in the final array has 6.




Input:

 m = 1

       n = 4

       array1 = [0]

       array2 = [1,4,8,10]
       k = 2
```

Output:

```
 4
```

Explanation:

```
 Merging both arrays and sorted. Final array will be -

 [1,4,8,10]

We can see at k = 2 in the final array has 4
```

Solution:

***Disclaimer**: Don't jump directly to the solution, try it out yourself first.*

Solution 1: Naive Solution

Intuition:

It is given that both arrays are sorted. We need to kth element which will be present when both are merged in a sorted manner. It gives us hints of approaching a solution with merge sort. Why so? If we see an algorithm of merge sort. It includes the following steps.

1. Divide the array into two halves.
2. Merge them in a sorted way.

So, we can use the method of merging two sorted arrays.

Approach :

We will keep two pointers, say p1 and p2, each in two arrays. A counter to keep track of whether we have reached the kth position. Start iterating through both arrays. If array1[p1] < array2[p2], move p1 pointer ahead and increase counter value. If array2[p2] <array1[p1],

move p2 pointer ahead and increase counter. When the count is equal to k, return the element in which condition makes the counter value equal to k.

Code:

- C++ Code
- Java Code
- Python Code

```
#include<iostream>
using namespace std;

int kthelement(int array1[],int array2[],int m,int n,int k) {
    int p1=0,p2=0,counter=0,answer=0;

    while(p1<m && p2<n) {
        if(counter == k) break;
        else if(array1[p1]<array2[p2]) {
            answer = array1[p1];
            ++p1;
        }
        else {
            answer = array2[p2];
            ++p2;
        }
        ++counter;
    }
    if(counter != k) {
        if(p1 != m-1)
            answer = array1[k-counter];
        else
            answer = array2[k-counter];
    }
    return answer;
}
```

```
int main() {
    int array1[] = {2,3,6,7,9};
    int array2[] = {1,4,8,10};
    int m = sizeof(array1)/sizeof(array1[0]);
    int n = sizeof(array2)/sizeof(array2[0]);
    int k = 5;
    cout<<"The element at the kth position in the final sorted array is "
    <<kthelement(array1,array2,m,n,k);
    return 0;
}
```

Output: The element at the kth position in the final sorted array is 6

Time Complexity :

We iterate at total k times. This makes time complexity to O(k)

Space Complexity :

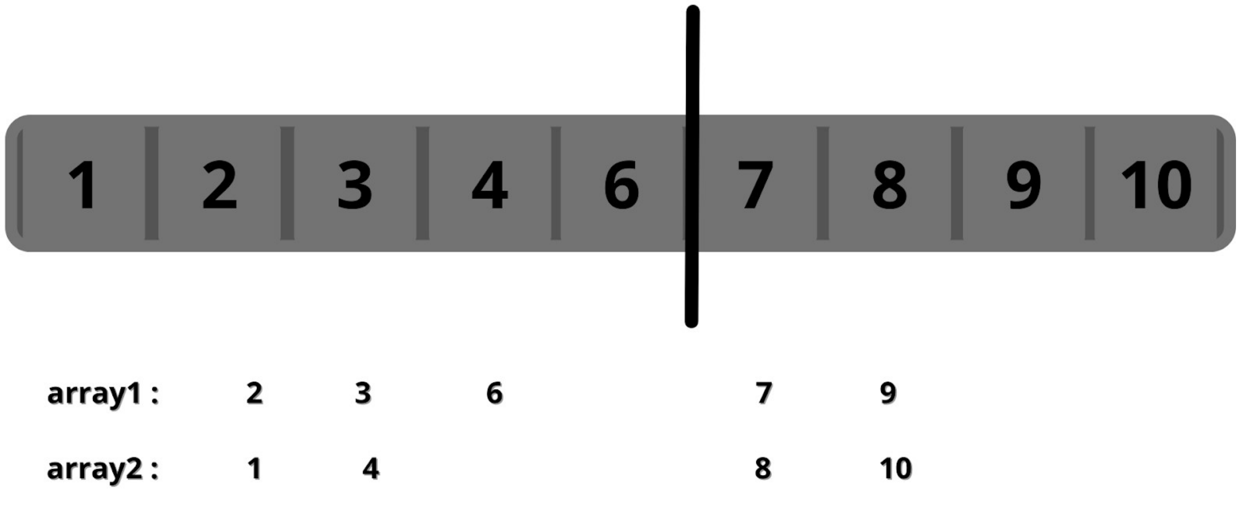
We do not use any extra data structure and hence, the time complexity is O(1).

Approach 2: Optimal Solution

Intuition :

It is mentioned that given arrays are sorted. This gives us some hints to use binary search in them.

If we look into the final merged sorted array.



We can part it in such a way that our kth element will be at the end of the left half array. Let's make some observations. The left portion of the array is made of some elements of both array1 and array2. We can see that all elements of the right half of the array are always larger than the left ones. So, with help of binary search, we will divide arrays into partitions with keeping k elements in the left half. We have to keep in mind that l1 <= r2 and l2 <= r1. Why so? This ensures that left-half elements are always lesser than right elements.

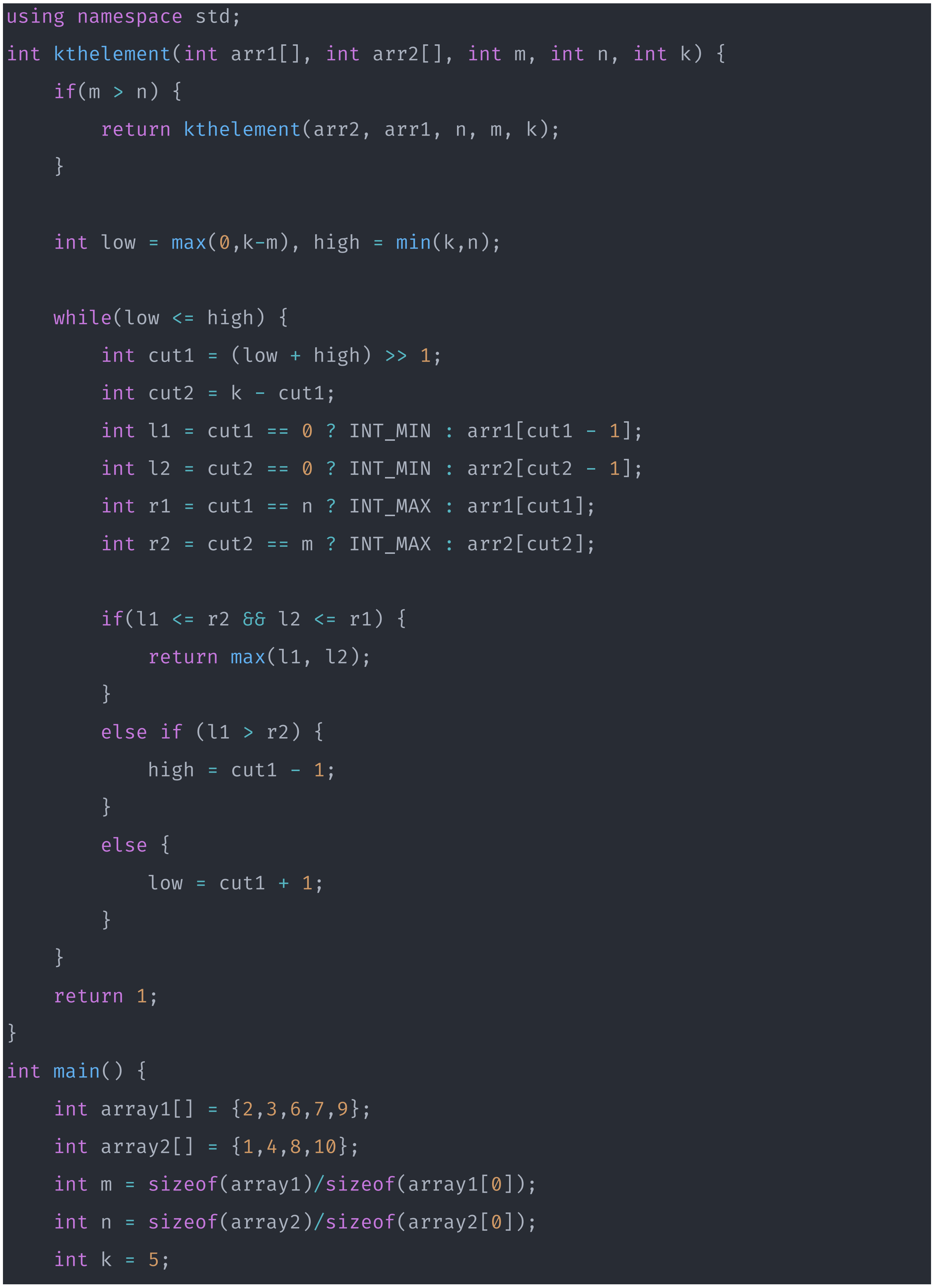
Approach :

Apply binary search in an array with a small size. Start iterating with two pointers, say left and right. Find the middle of the range. Take elements from low to middle of array1 and the remaining elements from the second array. Then using the condition mentioned above, check if the left half is valid. If valid, print the maximum of both array's last element. If not, move the range towards the right if l2 > r1, else move the range towards the left if l1 > r2.

Code:

- C++ Code
- Java Code
- Python Code

```
#include<bits/stdc++.h>
```



```
using namespace std;
int kthelement(int arr1[], int arr2[], int m, int n, int k) {
    if(m > n) {
        return kthelement(arr2, arr1, n, m, k);
    }

    int low = max(0,k-m), high = min(k,n);

    while(low <= high) {
        int cut1 = (low + high) >> 1;
        int cut2 = k - cut1;
        int l1 = cut1 == 0 ? INT_MIN : arr1[cut1 - 1];
        int l2 = cut2 == 0 ? INT_MIN : arr2[cut2 - 1];
        int r1 = cut1 == n ? INT_MAX : arr1[cut1];
        int r2 = cut2 == m ? INT_MAX : arr2[cut2];

        if(l1 <= r2 && l2 <= r1) {
            return max(l1, l2);
        }
        else if (l1 > r2) {
            high = cut1 - 1;
        }
        else {
            low = cut1 + 1;
        }
    }
    return 1;
}
int main() {
    int array1[] = {2,3,6,7,9};
    int array2[] = {1,4,8,10};
    int m = sizeof(array1)/sizeof(array1[0]);
    int n = sizeof(array2)/sizeof(array2[0]);
    int k = 5;
    cout<<"The element at the kth position in the final sorted array is "
    <<kthelement(array1,array2,m,n,k);
    return 0;
}
```

Output: The element at the kth position in the final sorted array is 6

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

Reason: We are applying binary search in the array with minimum size among the two. And we know the time complexity of the binary search is log(N) where N is the size of the array. Thus, the time complexity of this approach is log(min(m,n)), where m,n are the sizes of two arrays.

Space Complexity: O(1)

Reason: Since no extra data structure is used, making space complexity to O(1).