# An algorithm to find the location of kth sorted element in from two sorted arrays

Harsh Meena - IIT20190005,
Asha Jyothi Donga - IIT2019006,
Sampada Kathar - IIT2019007

Date: 10-02-2021

**Abstract**

In this paper we have devised an algorithm which finds the kth sorted element when two given sorted arrays are merged.

## 1 Problem

Given two sorted arrays of size m and n respectively, you are tasked with finding the element that would be at the k'th position of the final sorted array.

## 2 Keywords

Arrays, Sorting, Divide and Conquer.

## 3 Introduction

Divide And Conquer This technique can be divided into the following three parts:

Divide- This involves dividing the problem into some sub problem. Conquer- Sub problem by calling recursively until sub problem solved. Combine- The Sub problem Solved so that we will get find problem solution.

## 4 Algorithm 1 Design

To find the kth sorted element when two given sorted arrays are merged:

1. Compare the middle elements of arrays arr1 and arr2, let us call these indices mid1 and mid2 respectively.
2. Let us assume arr1[mid1] k, then clearly the elements after mid2 cannot be the required element.
3. Set the last element of arr2 to be arr2[mid2].
    - In this way, define a new subproblem with half the size of one of the arrays.

## 5 Algorithm 1 Analysis

```
int kth(int *arr1,int *arr2,int *end1,int
    *end2,int k)

   if arr1==end1
      return arr2[k]
   if arr2==end2
      return arr1[k]

   int mid1=(end1-arr1) / 2
   int mid2=(end2-arr2) / 2

   if mid1+mid2 < k
      if arr1[mid1] > arr2[mid2]
```

```
            return kth(arr1, arr2+mid2+1,
                end1, end2,
                k-mid2-1)
        else
            return kth(arr1+mid1+1, arr2,
                end1, end2,
                k-mid1-1)

    else
        if arr1[mid1] > arr2[mid2]
            return kth(arr1, arr2, arr1+mid1,
                end2, k)
        else
            return kth(arr1, arr2, end1,
                arr2+mid2, k)

}
```

### 5.1 Time Complexity Analysis

Using this recursive divide and conquer approach, the function is called a total of $logm + logn$ times. Thus the worst case complexity of this algorithm will be $O(logm + logn)$.
The best case occurs when either m or n is equal to 1, i.e., one of the arrays contains only a single element. Thus, the best case time complexity is $\Omega(1)$

### 5.2 Space Complexity

This algorithm has a space complexity of $O(logm + logn)$

## 6 Algorithm 2 Design

To find the index of the kth greatest element for the given Binary Max heap Array, we will use the same above algorithm and instead of dividing the segment in m/2 and n/2 and recursing again, we divide them both by k/2 and then recurse.

## 7 Algorithm 2 Analysis

```
int Kth(int arr1[],int arr2[],int m,int n,
    int k,int in1,int in2)
{
   int current = k/2
   if in1==m
      return arr2[in2+k-1]

   if in2==n
      return arr1[in1+k-1]

   if k==0 or k>((m-in1)+(n-in2))
      return -1

   if k==1
      int x=arr1[in1]<arr2[in2]?
          arr1[in1]:arr2[in2]
      return x

   if current-1 >= m-in1
      if arr1[m-1]<arr2[in2+current-1]
        return arr2[in2+(k-(m-in1)-1)]
      else
        return kth(arr1,arr2,m,n,k-current,
        in1,in2+current);

   if current-1 >= n-in2
      if arr2[n-1]<arr1[in1+current-1]
        return arr1[in1+(k-(n-in2)-1)]
      else
        return kth(arr1,arr2,m,n,k-current,
        in1+current,in2)
   }else{
      if arr1[current+in1-1]<arr2[in2+
                                current-1]
        return kth(arr1,arr2,m,n,k-current,
        in1+current,in2)
      else
        return kth(arr1,arr2,m,n,k-current,
        in1,in2+current)
   }
}
```

### 7.1 Time Complexity Analysis

Using this recursive divide and conquer approach, the function is called a total of $logk$ times. Thus the worst case complexity of this algorithm will be $O(logk)$.
The best case occurs when either m or n is equal to 0, or k is equal to 0. Thus, the best case time complexity is $\Omega(1)$

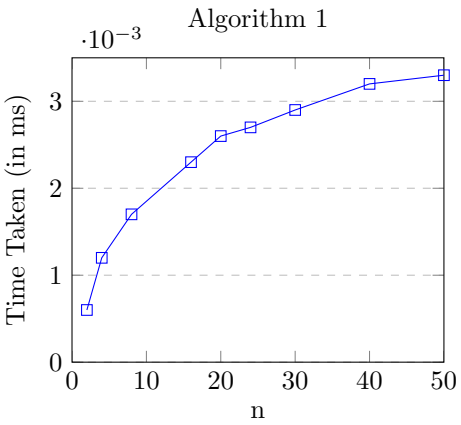
### 7.2 Space Complexity

This algorithm has a space complexity of $O(logk)$

## 8 Experimental Analysis

In the following table some cases are plotted for the first algorithm,

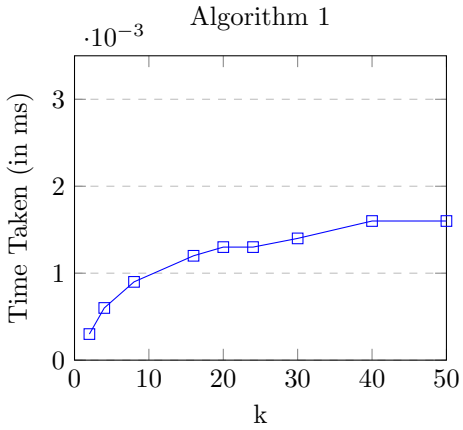
| m | n | Time Taken (in ms) |
|---|---|---|
| 2 | 2 | 0.0006 |
| 4 | 4 | 0.0012 |
| 8 | 8 | 0.0017 |
| 16 | 16 | 0.0023 |
| 20 | 20 | 0.0026 |
| 24 | 24 | 0.0027 |
| 30 | 30 | 0.0029 |
| 40 | 40 | 0.0032 |
| 50 | 50 | 0.0033 |




The above graph shows the variance of Time with respect to size of the second array, and follows our theoretically calculated TC of $O(log(m) + log(n))$.

In the following table some cases are plotted for the second algorithm,

| k | Time Taken (in ms) |
|---|---|
| 2 | 0.0003 |
| 4 | 0.0006 |
| 8 | 0.0009 |
| 16 | 0.0012 |
| 20 | 0.0013 |
| 24 | 0.0013 |
| 30 | 0.0014 |
| 40 | 0.0016 |
| 50 | 0.0016 |




The above graph shows the variance of Time with respect to the value of k, and follows our theoretically calculated TC of $O(log(k))$.

## 9 Conclusion

From this paper we can conclude that the second algorithm devised by us is the more efficient one for our purposes.
It is much more efficient than our first algorithm in terms of both time and memory, with a worst case time complexity of only O(log(k)) and a space complexity of O(log(k)).

## 10 References

[1]vaibhav29498, 'K-th Greatest Element in a Max-Heap', GeeksforGeeks, 2018. [Online]. [Accessed: 1-Feb-2021]
[2]'Heap Data Structures', tutorialspoint. [Online]. [Accessed: 1-Feb-2021]

[3]GeeksforGeeks, 'Priority Queue — Set 1 (Introduction)', GeeksforGeeks, 2018. [Online]. [Accessed: 1-Feb-2021]