An Algorithm to find the element present at k'th position in two sorted arrays.

DAA ASSIGNMENT-1 , GROUP 2

Vasu Gupta
IIB2019003

Saloni Singla
IIB2019004

Sandeep Kumar
IIB2019005

***Abstract*—This Paper contains the algorithm for finding the element that would be at the k'th position of the final sorted array generated from given two arrays of size m and n.**

I. Problem Statement

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.

II. Introduction

Let's first formally define what Divide and conquer algorithm is.
The Divide and conquer strategy solves a problem by:

1) Divide: Breaking the problem into sub problems that are themselves smaller instances of the same type of problem.
2) Recursion: Recursively solving these sub-problems.
3) Conquer: Appropriately combining their answers.

III. Algorithmic Design

A. *Approach 1*

1) If the value of k is greater than sum of n and m or k is less than 1 then return -1.
2) If one of the array is empty then return the element present at $(k-1)^{th}$ position in the other array.
3) If value of k is equal to 1 then return the minimum among the arr1[0] and arr2[0].
4) Store the minimum among n and k/2 & m and k/2 in i and j respectively (we are comparing size of arrays with k/2 to avoid Segmentation Fault if k/2 is greater than size of array).
5) In this divide and conquer approach we will divide both the arrays arr1[] and arr2[] by half of the value of k and then recurse the function.

B. *Approach 2*

1) If arr1 is equal to end1 then return the element present at $(k)^{th}$ in arr2[].
2) If arr2 is equal to end1 then return the element present at $(k)^{th}$ in arr1[].
3) Calculate the mid-points of the arr1[] and arr2[].
4) Lets make an assumption that element present at mid-point of arr1 is less than k then it is clear that the elements after mid-point of arr2[] cannot be the $(k)^{th}$ element.
5) So according to our assumption we will set the element present at mid-point of arr2[] as the last element of the arr2[].
6) In this way,we will get a new sub-problem with half the size of one of the arrays.

Algorithm 1: To find the $k'th$ element in the final sorted array

---

Input: Two arrays arr1 and arr2 of size n and m respectively and k such that $0 < k < m+n+1$
Output: Return the element at the $k'th$ position

Function `Kth`($arr1$,$arr2$,m,n,k)**:**
  if $k > m+n$ *or* $k < 1$ **then**
    return -1
  if $m = 0$ *and* $n > 0$ **then**
    return $arr2[k-m-1]$
  if $n = 0$ *and* $m > 0$ **then**
    return *arr1[k-n-1]*
  if $k = 0$ **then**
    return *minimum(arr1[0],arr2[0])*
  $i \leftarrow minimum(m, k/2)$ $j \leftarrow minimum(n, k/2)$
  if $arr1[i-1] < arr2[j-1]$ **then**
    return $kth(arr1+i, arr2,\ m-i\ ,\ n, k-i)$
  else
    return $kth(arr1, ar2+j, m,\ n-j,\ k-j)$

Algorithm 2: To find the $k'th$ element in the final sorted array

Input: Two arrays arr1 and arr2 of size n and m respectively and k such that $0 < k < m + n + 1$
Output: Return the element at the $k'th$ position

Function Kth($array1$,$array2$,$end1$,$end2$,$k-1$):
    if *array1* = *end1* **then**
        return $array2[k]$
    if *array2* = *end2* **then**
        return *array1[k]*
    $mid1 \leftarrow (end1 - array1)/2$
    $mid2 \leftarrow (end2 - array2)/2$
    if *mid1+mid2* $< k$ **then**
        if *array1[mid1* $>$ *array2[mid2]* **then**
            return *kth*($array1$,$array2 + mid2 + 1$, $end1$,$end2$,$k - mid2 - 1$)
        else
            return *kth*($array1 + mid1 + 1$,$array2$, $end1$,$end2 + mid2$,$k - mid1 - 1$)
    else
        if *array1[mid1]* $>$ *array2[mid2]* **then**
            return *kth*($array1$,$array2$, $array1 + mid1$,$end2$,k)
        else
            return *kth*($array1$,$array2$, $end1$,$array2 + mid2$,k)

IV. Algorithm Analysis

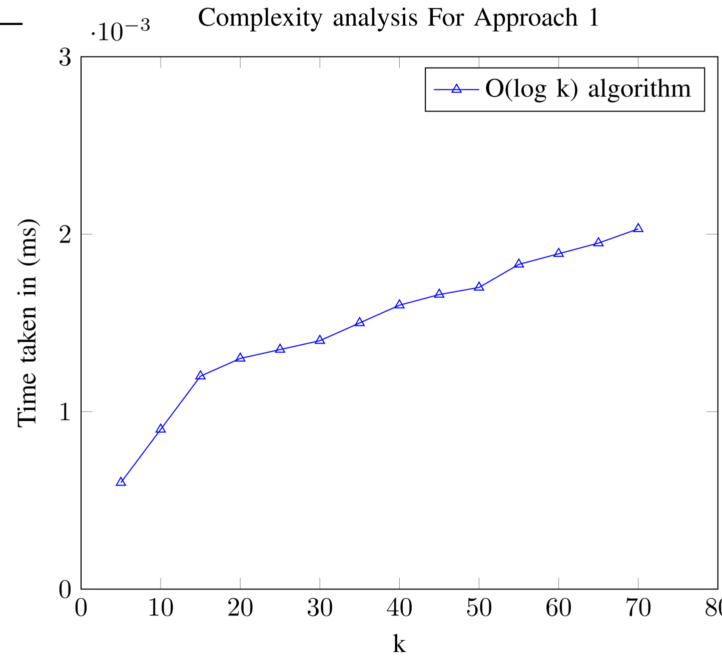
A. *Approach 1*

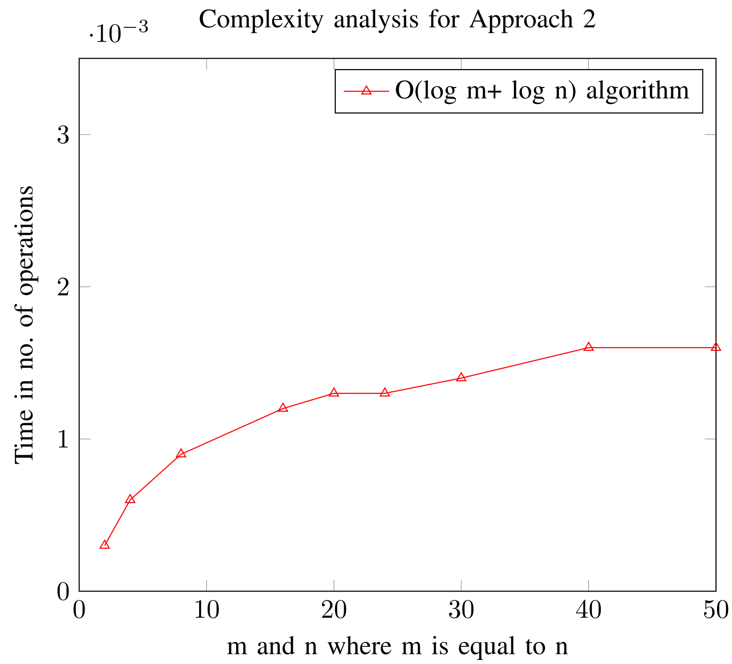
Time Complexity Analysis
In this recursive divide and conquer approach, the function kth is called a total of logm+logn times. Thus the time complexity of this approach would be $O(\log k)$. The best case complexity will be when either of m and n is zero or k is invalid or k is equal to 1. Thus, the best case time complexity is $\Omega(1)$
Space Complexity Analysis
This algorithm has a space complexity of O(log k)

B. *Approach 2*

Time Complexity Analysis
In this recursive divide and conquer approach, the function kth is called log k times. Thus the time complexity of this approach would be $O(\log m + \log n)$.
The best case complexity will be when either of m and n is zero or k is invalid or k is equal to 1. Thus, the best case time complexity is $\Omega(1)$
Space Complexity Analysis
This algorithm has a space complexity of O(log m+log n)

V. Experimental Study





VI. Conclusion

Above two methods have different time complexities and meet to fulfill the problem statement. The order in which they are good can be listed as:
I. Approach 1
II. Approach 2
Based on the time complexities.

VII. References

https://www.geeksforgeeks.org/k-th-element-two-sorted-arrays/

https://tutorialspoint.dev/algorithm/divide-and-conquer/k-th-element-two-sorted-arrays