# Theory Assignment-1: ADA Winter-2024

Aarzoo (2022008) Anushka Srivastava (2022086)

## 1 Preprocessing

Before arriving at a final solution, we approached the question step by step, as instructed by the $Q21$ of the book **"Algorithms by Jeff Erickson."** (https://jeffe.cs.illinois.edu/teaching/algorithms/book/01-recursion.pdf)

For the first step, we came up with a solution to find the median element of the union of two sorted arrays, $A$ and $B$, of the same size n in $\Theta(\log(n))$ time, for the $(a)$ part of the question.

For the second step, we came up with a solution to find the $kth$ smallest element of two sorted arrays, A and B, of different sizes $m$ and $n$ in $\Theta(\log(m+n))$ time, on the basis of our solution to the first part, for the $(b)$ part of the question.

For our final solution, we used the logic of our approach from the second step to modify the function to handle the same problem in the case of three arrays, the algorithm for which is described in detail in the next section.

**Preprocessing for the input:** The code calculates the value of $kth$ smallest element by assuming that when $k = 0$, the smallest element will be returned. So, before passing the value of $k$ in the initial function call, the value of $k$ is reduced by 1. For example, when $k = 1$, when the smallest element needs to be computed, the value of $k$ is reduced to 0 for further computation.

## 2 Algorithm Description

The algorithm is based on the Divide and Conquer rule.

- **Base Case:** The base case involves the condition when the length of one of the arrays becomes 0 thus arriving at the case of finding the kth smallest element in the union of 2 sorted arrays which may have varying sizes. A separate function handles this case.
- Description of the function for finding the kth element of 2 sorted arrays :

  **Base Case:** This function begins with the base case if the length of one of the arrays is 0, then simply return the kth smallest from the second array.

  **Recursive Step:** The algorithm compares the sum of the mid indexes of the two arrays with k.

  **Case 1:** If the sum is smaller than k, it checks which middle element is greater. If the middle element of the first array is greater, the function is recursively called for the first array and the second half of the second array.

  **Case 2:** If the sum is greater than k, it again checks for the greater middle element. For instance, if the mid of the first array is greater than the second, the function is recursively called for the first half of the first array and the second array.
- Recursive Step for the main function:

  The main function extends the recursive step to handle three sorted arrays:

  **Case 1:** If the sum of mid indexes of all three arrays is greater than k, it checks for the smallest among the middle elements of the three arrays. Then, it recursively calls the function to find the (k - mid - 1) element from the second half of the array containing the smallest middle element and the other two arrays.

  **Case 2:** If the sum is less than k, it checks for the greatest among the middle elements of the three arrays. Then, it recursively calls the function for the first half of that array and the other two arrays.

# 3 Recurrence Relation

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

In every recursion call, the size of only one of the arrays is being halved. So, $T(n)$ is reduced to $T(\frac{n}{2})$ in every step of the recurrence relation. In rest of the computations, it takes $O(1)$ time.

# 4 Complexity Analysis

The time complexity of the algorithm is $O(\log(n))$ and the space complexity of the algorithm is $O(\log(n))$.

- **Space Complexity:** The space complexity of the algorithm will be determined by the maximum depth of the recursion. In the algorithm, the maximum depth of recursion is log(min(p, q, r)), where p, q and r are the size of the arrays. And the array length of only one array is being halved with each recursive call. Therefore, the space complexity of the algorithm would be O(log(min(p, q, r))) which is eventually $O(\log(n))$ since the initial size of each array is equal to n.
- **Time Complexity:** At each level of the recursion tree, we have O(1) work done. The height of the recursion tree is log(min(p, q, r)), as the array length of exactly one array is halved in each recursive call. Therefore, the total work done by the algorithm is O(log(min(p, q, r))). So, the final time complexity of the function, taking into account the correct base case, is O(log(min(p, q, r))) which is eventually O(log(n)).

$$T(p,q,r)$$
$$\downarrow$$
$$One-of: T\left(\frac{p}{2},q,r\right) \quad T\left(p,\frac{q}{2},r\right) \quad T\left(p,q,\frac{r}{2}\right)$$

As shown above, the recursive steps choose one of the paths and eventually reach the base case of finding the kth smallest element from 2 arrays. Thus the time complexity is O(log(n)).

# 5 References


- **Algorithms** by Jeff Erickson
- **Stack Overflow** - kth smallest element in the union of two sorted arrays
- **TakeUForward** - Median of row wise sorted matrix


# 6 Pseudocode

## 6.1 Assumptions Taken

- Our algorithm uses 0-based indexing.
- Initially, we are starting with three equal-sized arrays of size $n$. On further computation, the lengths of the arrays are bound to change, so we take three general variables $p$, $q$, and $r$ to represent the lengths of the three arrays.
- **For Alternate Solution:** Adding elements in an array by simply assigning it as an element takes $O(1)$ time.

**Algorithm 1** Main solution: Using Divide and Conquer

```
function kthSmallestin2Arrays(arr1, arr2, k)
    if len(arr1) = 0 then
        return arr2[k]
    else if len(arr2) = 0 then
        return arr1[k]
    end if
    mid1 ← len(arr1)/2
    mid2 ← len(arr2)/2
    if mid1 + mid2 < k then
        if arr1[mid1] > arr2[mid2] then
            return kthSmallestin2Arrays(arr1, arr2[mid2 + 1 :], k − mid2 − 1)
        else
            return kthSmallestin2Arrays(arr1[mid1 + 1 :], arr2, k − mid1 − 1)
        end if
    else
        if arr1[mid1] > arr2[mid2] then
            return kthSmallestin2Arrays(arr1[: mid1], arr2, k)
        else
            return kthSmallestin2Arrays(arr1, arr2[: mid2], k)
        end if
    end if
end function

function kthSmallest(arr1, arr2, arr3, k)
    if len(arr1) = 0 then
        return kthSmallestin2Arrays(arr2, arr3, k)
    else if len(arr2) = 0 then
        return kthSmallestin2Arrays(arr1, arr3, k)
    else if len(arr3) = 0 then
        return kthSmallestin2Arrays(arr1, arr2, k)
    end if
    mid1 ← len(arr1)/2
    mid2 ← len(arr2)/2
    mid3 ← len(arr3)/2
    if mid1 + mid2 + mid3 < k then
        if arr1[mid1] > arr2[mid2] and arr1[mid1] > arr3[mid3] then
            if arr2[mid2] < arr3[mid3] then
                return kthSmallest(arr1, arr2[mid2 :], arr3, k − mid2 − 1)
            else
                return kthSmallest(arr1, arr2, arr3[mid3 :], k − mid3 − 1)
            end if
        else if arr2[mid2] > arr1[mid1] and arr2[mid2] > arr3[mid3] then
            if arr1[mid1] > arr3[mid3] then
                return kthSmallest(arr1, arr2, arr3[mid3 :], k − mid3 − 1)
            else
                return kthSmallest(arr1[mid1 :], arr2, arr3, k − mid1 − 1)
            end if
        else
            if arr1[mid1] < arr2[mid2] then
                return kthSmallest(arr1[mid1 + 1 :], arr2, arr3, k − mid1 − 1)
            else
                return kthSmallest(arr1, arr2[mid2 + 1 :], arr3, k − mid2 − 1)
            end if
        end if
    else
        if arr1[mid1] > arr2[mid2] and arr1[mid1] > arr3[mid3] then
            return kthSmallest(arr1[: mid1], arr2, arr3, k)
        else if arr2[mid2] > arr1[mid1] and arr2[mid2] > arr3[mid3] then
            return kthSmallest(arr1, arr2[: mid2], arr3, k)
        else
            return kthSmallest(arr1, arr2, arr3, arr3[: mid3], k)
        end if
    end if
end function
```

**Algorithm 2** Alternate Solution: Using Binary Search

```
function countSmallerThanMid(row, mid)
    l ← 0
    h ← len(row) − 1
    while l ≤ h do
        md ← (l + h)/2
        if row[md] ≤ mid then
            l ← md + 1
        else
            h ← md − 1
        end if
    end while
    return l
end function

function kthSmallest(arr1, arr2, arr3, k)
    A ← [arr1, arr2, arr3]
    low ← 1
    high ← 1e9
    n ← 3
    m ← len(arr1)
    while low ≤ high do
        mid ← (low + high)/2
        count ← 0
        for i ← 0 to n − 1 do
            count ← count + countSmallerThanMid(A[i], mid)
        end for
        if count < k then
            low ← mid + 1
        else
            high ← mid − 1
        end if
    end while
    return low
end function
```

**Reference and Explanation for the Algorithm 2 : Alternate solution: Link for the solution**

- The solution is based on finding the kth smallest element by creating a matrix of size 3 X n. The algorithm is based on nested binary searches. One binary search is to find the kth smallest element whereas the other is used to find the count of the number of elements smaller than and equal to the given element. The time complexity of the algorithm is $O(\log(n))$.

# 7 Proof of Correctness

**Idea:** Check the sum of mid indexes and compare with k to eliminate those halves of arrays that cannot contain the kth smallest element. This process continues recursively until the length of any of the three arrays reaches zero. The kth smallest element can then be checked in the remaining two arrays through a separate function, which recursively calls itself using a similar logic until the length of one of the arrays reaches 0. Following this, the kth smallest element of the remaining array is simply returned.

**kthLargestin2Arrays**

**Case 1:** When $k >$ mid1 + mid2

- mid1 + mid2 = $(p + q)/2$, where $p$ and $q$ are lengths of the arrays after $n$ recursive calls of the function.
  So, intuitively, if $k$ is greater than half the total length of both arrays, we can definitely say that the kth smallest element doesn't lie in the first half of the array where arr[mid] is lower than the other array.
  In the recursive call, we eliminate the left half of the array with a lower median and adjust the value of $k$ by decreasing it by the median's index to find the new kth smallest element in the remaining array elements.

**Case 2:** When $k <=$ mid1 + mid2

- If $k$ is less than or equal to the half of the total length of both arrays, we can say that the kth smallest element definitely doesn't lie in the second half of the array where arr[mid] is greater.
  In the recursive call, the right half of that particular array is eliminated, and the value of $k$ is passed as it is because the removed elements come after the kth element.

This process continues until any of the arrays becomes empty. We have eliminated all the elements that don't contain the kth smallest element for the original value of $k$. So, the kth smallest element (after adjusting the value of $k$) is simply the kth element of the remaining array.

**kthLargest (Main subroutine)**

**Case 1:** When $k >$ mid1 + mid2 + mid3

- mid1 + mid2 + mid3 = $(p + q + r)/2$, where $p$, $q$, and $r$ are lengths of the three arrays after $n$ recursive calls.
  So, intuitively, if $k$ is greater than half the total length of all the arrays, we can definitely say that the kth smallest element doesn't lie in the first half of the array where arr[mid] is lower than the arr[mid] of the rest of the arrays.
  In the recursive call, we eliminate the left half of the array with a lower median and adjust the value of $k$ by decreasing it by the median's index to find the new kth smallest element in the remaining array elements.

**Case 2:** When $k <=$ mid1 + mid2 + mid3

- If $k$ is less than or equal to half the total length of all the arrays, we can say that the kth smallest element definitely doesn't lie in the second half of the array where arr[mid] is the greatest out of all arrays.
  In the recursive call, the right half of that particular array is eliminated, and the value of $k$ is passed as it is because the removed elements come after the kth element.

This process continues until any of the arrays becomes empty. As the base case, we compute the kth smallest element (after adjusting the value of $k$) in the remaining two arrays, which is then handled by the **kthLargestin2Arrays** function. The kth element is then returned after eliminating all the elements using divide and conquer.