# Theory Assignment-1: ADA Winter-2024

Niteen Kumar (2022336) Shubham (2022488)

## 1 Assumptions

0-based indexing, array indices start from 0. All arrays have the same number of elements, $n$. Also, $k \in (0, 3n-1)$.

Operations like arithmetic (addition, division, subtraction, multiplication), comparison between two elements take constant time.

We are assuming that there will be no overflows.

## 2 Algorithm Description

The algorithm aims to find the $k$-th smallest element among three sorted arrays $a$, $b$, and $c$. We do a binary search sort of algorithm to find the $k$-th smallest element. Suppose we start with array $a$. We run a binary search on it to find an element which has $k-1$ smaller elements than itself. We can use a custom function to find the number of smaller elements (count) than a given element (key) in all three arrays and compare with $k$ and update the range accordingly.

$$\text{count} = \text{smaller elements than key in array } a + \text{smaller elements than key in array } b + \text{smaller elements than key in array } c$$

If $count == k$: key is the answer.
if $count > k-1$: we need to search in the right side of key.
if $count < k-1$: we need to search in the left side of key.

Now if we don't find the $k$-th smallest element in array $a$, we can try to find the element in array $b$ and $c$ using the same steps described above.

If the arrays contain duplicate elements we can just take the range of key.

$$\text{count1} = \text{smaller elements than key in array } a + \text{smaller elements than key in array } b + \text{smaller elements than key in array } c$$

$$\text{count2} = \text{elements smaller and equal to key in array } a + \text{elements smaller and equal to key in array } b + \text{elements smaller and eq}$$

(We don't count the element while counting the equal elements)
if $count1 < k$ **and** $k \leq count2$ : key is the answer.
if $count1 >= k$: search in the left side of key.
else search in the right side of the key.

Now of course, we will find the $k$-th element in one of the arrays, as $0 \leq k < 3n$.

## 3 Recurrence Relation

Iterative solution , so no recurrence relation

## 4 Time Complexity Analysis

### 4.1 Finding the Specific Element

The process of repeatedly dividing the search space in half efficiently finds an element with $k-1$ smaller elements. Each iteration runs a logarithmic number of times, halving the search space, leading to $O(\log n)$ time complexity.

### 4.2 Counting Elements within the Search

Within each iteration of the above process, a method is used to count elements smaller than a given value in each array. It uses lower bound and upper bound function which takes $O(2 \log n)$ times .

### 4.3 Handling Different Array Orders

In the worst case, the code tries different orders of the arrays to ensure the desired element isn't missed. However, this only adds a constant factor overhead, not affecting the dominant growth rate and takes 3 iterations at max

### 4.4 Overall Time Complexity

The nested logarithmic behavior of the primary search process and the counting method within it leads to $O(\log n \cdot 6 \log n) = O(6 \log^2 n)$ time complexity.

## 5 Space Complexity Analysis

### 5.1 Auxiliary Space

The code uses a constant amount of auxiliary space, independent of the size of the input arrays. The space required for variables like `int l`, `int r`, `int mid`, and other local variables used in the functions is constant and does not depend on the input size. Therefore, the auxiliary space complexity is $O(1)$.

### 5.2 Input Space

The input space refers to the space required to store the input data (arrays $a$, $b$, and $c$). The space required for the input arrays is $O(n)$, where $n$ is the size of each array. The input space complexity is determined by the size of the input arrays.

### 5.3 Overall Space Complexity

The overall space complexity is the sum of the auxiliary space and input space complexities: $O(1)$ (auxiliary space) + $O(n)$ (input space) = $O(n)$. Therefore, the overall space complexity of the provided code is $O(n)$.

## 6 Pseudo code

**Algorithm 3** Function to find $k$-th smallest element among three sorted arrays

```
function CountSmallerAndEqualNumbers(arr, n, key)
    l = lower_bound(arr, n, key)
    r = upper_bound(arr, n, key)    ▷ returns the number of smaller elements and the number of smaller elements + number of equal elements
    return (l, r)
end function
```

```
function BinarySearchForAnswer(a, b, c, k, n)
    i = 0, j = n − 1
    while i ≤ j do
        mid = (i + j)/2
        range_a = CountSmallerAndEqualNumbers(a, n, a[mid])
        range_b = CountSmallerAndEqualNumbers(b, n, a[mid])
        range_c = CountSmallerAndEqualNumbers(c, n, a[mid])
        count1 = range_a.first + range_b.first + range_c.first
        count2 = range_a.second + range_b.second + range_c.second
        if count1 < k and k ≤ count2 then
            return a[mid]
        else if count1 ≥ k then
            j = mid − 1
        else
            i = mid + 1
        end if
    end while
     return -1
end function
```

```
function answer(n, k, a, b, c)
    ans = ok(a, b, c, k, n)
    if ans ≠ −1 then
        return ans
    else
        ans = ok(b, a, c, k, n)
        if ans ≠ −1 then
            return ans
        else
            ans = ok(c, b, a, k, n)
            return ans
        end if
    end if
end function
```

# 7 Proof of Correctness

The k-th smallest element in an array must have k-1 smaller or same elements to itself . The function CountSmallerAndEqualNumbers tries to find the total number of elements which are smaller or equal in all three arrays for a given element . Thus we can simply compare that number with k . Now as the arrays are sorted , the number of element smaller than , say x , is monotonic .

if x increases the number of elements smaller than or equal to it also increases , So we can use binary search to find the element which have exactly k-1 smaller or equal element to itself .

# 8 Example

## 8.1 Arrays

$$a = [1, 2, 3, 4, 5, 6, 7, 8, 9]$$
$$b = [2, 4, 6, 8, 10, 12, 14, 16, 18]$$
$$c = [3, 6, 9, 12, 15, 18, 21, 24, 27]$$

## 8.2 Target Index (k)

$k = 7$ (finding the 7th smallest element)

## 8.3 Steps

- The algorithm starts with array $a$ and performs binary search.
- i = 0 , j = 8(n-1)
- It checks the middle element, $a[4] = 5$.
- Using CountSmallerAndEqualElements , it finds:
  - 4 elements smaller than 5 in $a$
  - 2 elements smaller than 5 in $b$
  - 1elements smaller than 5 in $c$

  Total = 7 elements smaller than 5.
- Since 7 is greater than 6(k-1), the algorithm searches in the left subarray of $a$.
- i = 0 , j = 3
- It checks $a[1] = 2$.
- Using CountSmallerAndEqualElements , it finds:
  - 1 elements smaller than 2 in $a$
  - 1 element equal to 2 in $b$
  - 0 element smaller than 2 in $c$

  Total = 2 elements smaller or equal to 2.
- Since 2 is less than 6, the algorithm searches in the right subarray of $a$
- i = 2 , j = 3
- It checks $a[2] = 3$.
- Using CountSmallAndEqualElements , it finds:
  - 2 elements smaller than 3 in $a$

- 1 elements smaller than 3 in $b$
- 1 elements equal to 3 in $c$

Total = 4 elements smaller than or equal to 3

- Since 4 is smaller than 6 , the algorithm searches in the right subarray of $a$
- i = 3 , j = 3
- a[3] = 4
- Using CountSmallAndEqualElements , it finds:
  - 3 elements smaller than 4 in $a$
  - 2 elements smaller than or equal to 4 in $b$
  - 1 elements smaller than 4 in $c$
  - Since 6 is exactly the desired count (k-1), the algorithm returns 4 as the 7th smallest element.

Therefore, the algorithm correctly identifies 4 as the 7th smallest element across the three arrays.

Github repo : https://gitfront.io/r/Kniteenk/PUUZzqiGeP4x/ADA-assignments/