Theory Assignment-1: ADA Winter-2024

Shamik Sinha (2022468)   Vansh Yadav (2022559)

Algorithm Description

Assumptions Made

1. The arrays are composed of integers i.e. they are integer arrays or at the very least have numeric values (floating point or integer).
2. The array indexing follows a 0-based convention. An 'n' sized array has starting index 0 and last index at 'n'-1.
3. The arrays, as well as the elements within and between arrays, are distinct without any duplicates, ensuring uniqueness.

Input

Three sorted array of A, B and C integers (or floating point values).
An integer representing the size of each of the three arrays. It is assumed that all three arrays have the same size, denoted by 'n'.
An integer representing the desired rank of the element we want to find in $A \cup B \cup C$, denoted by 'k'.

Output

A single numeric value (Integer or Float), which is the 'k'th smallest value in $A \cup B \cup C$.

Solution Description

We decided to utilise the Binary Search algorithm to solve the problem. Using this technique, we divide the arrays into two parts using nested binary search and iteratively check whether we have a suitable partition where the k smallest elements of all the three arrays are on the left partition and the rest in the right partition.

The problem can be broken down into three parts as is the case with any problem which is solved with the Binary Search algorithm. We determine the 'low' and the 'high' values, calculate a corresponding middle value, check for a given condition on the obtained value and then finally, iterate or end the binary search depending on the conditions used.

1. Initialisation

The algorithm begins by initializing two pointers ('low1' and 'high1') for the first array A.
It also initializes a variable 'left' to represent the number of elements remaining to reach the k-th smallest element.

2. First Binary Search

Conducts a binary search within the range $[\max(0, k-2n), \min(k, n)]$ to find the potential position of the k-th smallest element in the first array A.

3. Second Binary Search

Inside the first binary search loop, a secondary binary search is performed for the second array B within a range determined by the remaining elements needed (left) to reach the k-th element.

4. Conditions and Comparisons

Within the second binary search loop, the algorithm checks various conditions involving midpoints and adjacent elements to determine the k-th smallest element.

Conditions consider comparisons between adjacent elements of the three arrays to identify the correct position of the k-th element.

5. Adjustment of Search Ranges

Based on the comparisons and conditions, the algorithm adjusts the binary search ranges for A and B accordingly to narrow down the search space.

6. Output

Once the k-th smallest element is found, the algorithm prints or returns the result.

Dry Run

As an example, we have shown below a dry run of our algorithm with the arrays $\{1, 2, 3, 4\}$, $\{5, 6, 7, 8\}$, and $\{9, 10, 11, 12\}$.

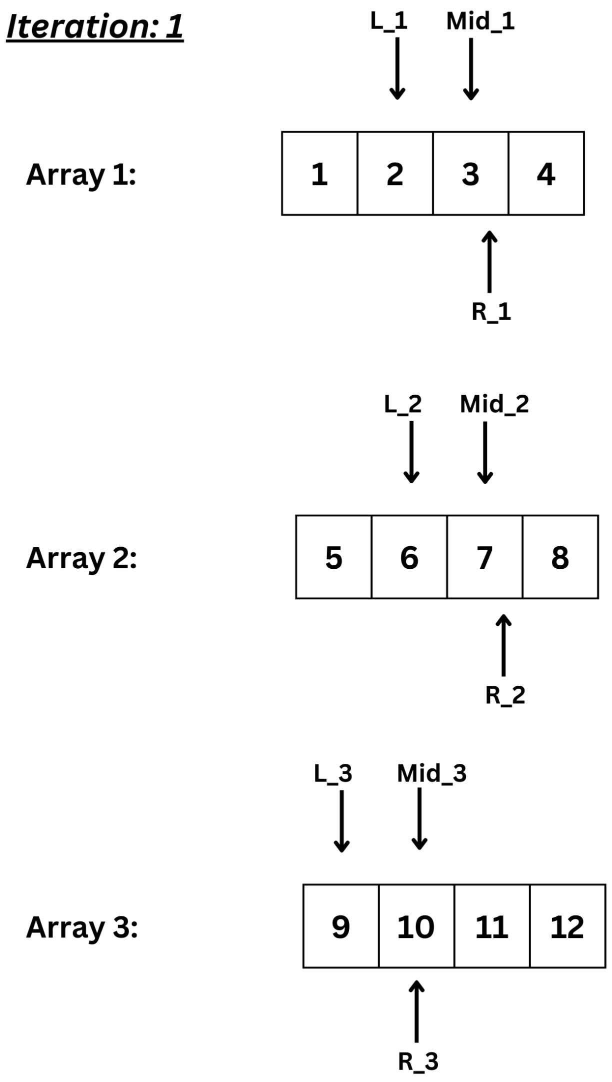

Figure 1: Iteration 1, in which $l2 > r1$ and $l3 > r1$, therefore, low1 is updated to mid1+1

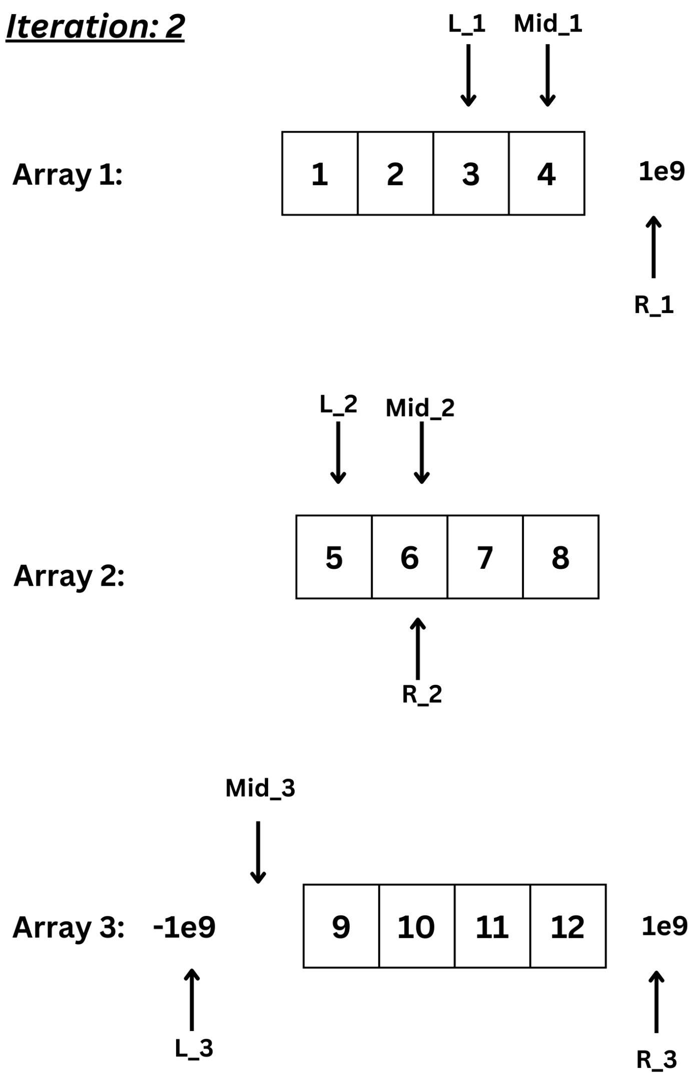

Figure 2: Iteration 2, in which $l2 > r1$, therefore, low1 is updated to mid1+1 again

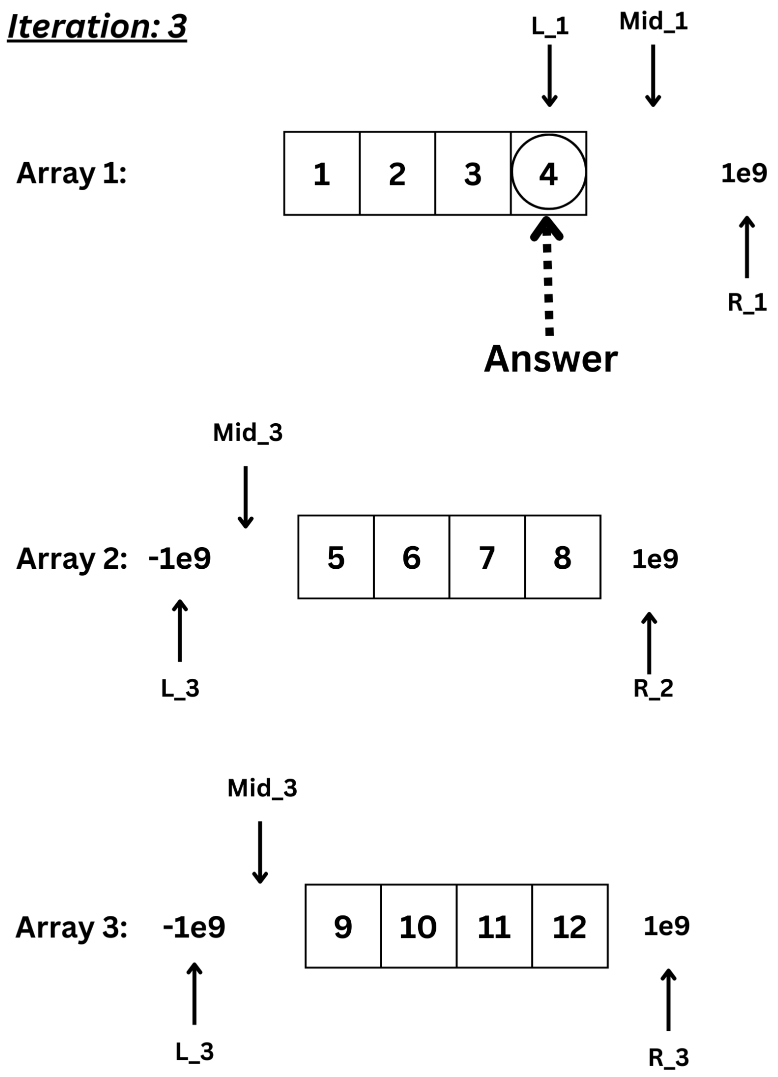

Figure 3: Iteration 3, in which all conditions are met.

Complexity Analysis

The code consists of two nested binary search loops, and within each binary search loop, there are constant-time operations. Let's denote the size of each array as 'n'.

Within the first binary search, with each iteration, the size of the array is reduced using the result of the previous comparison.

Initial length of array $A = n$

Iteration 1 - Length of array $A = \frac{n}{2}$

Iteration 2 - Length of array $A = \left(\frac{n}{2}\right) \div 2 = \frac{n}{2^2} = \frac{n}{4}$

Iteration k - Length of array $A = \frac{n}{2^k}$

After k iterations, the size of the array becomes 1 (narrowed down to the first element or last element only).

Length of array $A = \frac{n}{2^k} = 1 \implies 2^k = n$

Applying the logarithm function to both sides:

$$\log_2(n) = \log_2(2^k) \implies k = \log_2(n)$$

Therefore, $k = \log_2(n)$.

This implies the binary search algorithm runs in $\mathcal{O}(\log_2 n)$ time in the worst case.

Similarly, within the inner while loop executing binary search, with other time constant operations, it can be shown that it runs in logarithmic $\mathcal{O}(\log_2 n)$ time too.

Finally, since the second binary search is nested inside the first binary search, we multiply the time complexities. As a result, our program runs in $\mathcal{O}(\log_2^2 n)$ time in the worst case.

Pseudocode

Below is the pseudocode which represents the logic of the C++ code we had written for the problem, in a more abstract and readable form. The function takes three sorted arrays (A, B, C), their size (n), and the value of k (position of the desired element). The function returns the 'k'th smallest element among all the arrays.

Algorithm 1 An Algorithm to find the Kth smallest element in $A \cup B \cup C$, where A, B, and C are sorted integer arrays of size n.

```
function KthElement($A, B, C, n, k$):
    $low1 \leftarrow \max(0, k - 2 * n)$
    $high1 \leftarrow \min(k, n)$
    $left \leftarrow k$
    while $low1 \leq high1$ do
        $mid1 \leftarrow (low1 + high1)/2$
        $left1 \leftarrow left - mid1$
        $low2 \leftarrow \max(0, left1 - n)$
        $high2 \leftarrow \min(left1, n)$
        while $low2 \leq high2$ do
            $mid2 \leftarrow (low2 + high2)/2$
            $mid3 \leftarrow left1 - mid2$
            $r1 \leftarrow (A < n)\ ?\ A[mid1] : +\infty$
            $r2 \leftarrow (B < n)\ ?\ B[mid2] : +\infty$
            $r3 \leftarrow (C < n)\ ?\ C[mid3] : +\infty$
            $l1 \leftarrow (A - 1 \geq 0)\ ?\ arr1[A - 1] : -\infty$
            $l2 \leftarrow (B - 1 \geq 0)\ ?\ arr2[B - 1] : -\infty$
            $l3 \leftarrow (C - 1 \geq 0)\ ?\ arr3[C - 1] : -\infty$
            if $l1 \leq r2$ and $l2 \leq r1$ and $l2 \leq r3$ and $l3 \leq r2$ and $l1 \leq r3$ and $l3 \leq r1$ then
                print $\max(l1, \max(l2, l3))$
                return
            end if
            if $l1 > r2$ or $l1 > r3$ then
                $high1 \leftarrow mid1 - 1$
                break
            else if $x \leq n_0$ and $y > n_0$ then
                $low1 \leftarrow mid1 + 1$
                break
            else if $l2 > r1$ and $l2 > r3$ then
                if $r3 < r1$ then
                    $high2 \leftarrow mid2 - 1$
                else
                end if
            else if $l2 > r1$ and $l2 > r3$ then
                if $r3 < r1$ then
                    $high2 \leftarrow mid2 - 1$
                else
                    $low1 \leftarrow mid1 + 1$
                    break
                end if
            else if $l3 > r1$ and $l3 > r2$ then
                if $r1 < r2$ then
                    $low1 \leftarrow mid1 + 1$
                    break
                else
                    $low2 \leftarrow mid2 + 1$
                end if
            else if $l2 > r1$ or $l3 > r1$ then
                $low1 \leftarrow mid1 + 1$
                break
            else if $l2 > r3$ then
                $high2 \leftarrow mid2 - 1$
            else if $l3 > r2$ then
                $low2 \leftarrow mid2 + 1$
            end if
        end while
    end while
end function
```

Proof of Correctness

We can prove the correctness of our algorithm by demonstrating that it always produces the correct output for any valid input. Our algorithm which finds the k-th smallest element in the union of three sorted arrays, we can approach the proof of correctness using the principle of mathematical induction in the following steps:

Claim 1. *The algorithm is correct and gives the optimal solution. It gives the k-th smallest element in the union of three sorted n-sized arrays A, B and C.*

Proof.

Invariant

Firstly, identifying an invariant that holds true at the beginning and end of each iteration of the loops. Invariants are properties that are always true during the execution of the algorithm.

For our algorithm, At any point during the execution of the algorithm, the **k-th smallest element** is guaranteed to be within a certain range in the union of the three sorted arrays.

Base Case

This step involves proving that the algorithm works correctly for the base case, i.e., when the search space is small enough to be solved directly without recursion.

In our case, for the smallest input, where $\mathbf{n = 1}$ and $\mathbf{k = 1}$, the algorithm correctly identifies the k-th smallest element as the first element in the union of the arrays.

Inductive Step

In the inductive step we prove that if the algorithm correctly solves the problem for a certain input size, it also correctly solves the problem for the next larger input size.

Let us assume that our algorithm correctly finds the k-th smallest element for a given input size 'n'. In each iteration, the algorithm adjusts the search ranges based on comparisons and conditions, ensuring that the correct position of the k-th smallest element is maintained.

Termination

Lastly, we show that the algorithm terminates after a finite number of steps.

Our algorithm terminates when the search space is reduced to a small enough size. In each iteration, the search space is consistently reduced, moving closer to the base case. This guarantees termination after a finite number of steps.

Hence, the claim is proved. □

Bibliography

1. **GeeksforGeeks:** Binary Search – Data Structure and Algorithm Tutorials
2. **Scaler:** Time Complexity of Binary Search
3. **take U forward:** K-th element of two sorted arrays
4. **take U forward:** Median of two Sorted Arrays of Different Sizes
5. **CSE222:** Lecture Notes