Theory Assignment-1: ADA Winter-2024

Vivaswan Nawani (2021217) Animesh Pareek (2021131)

1 Assumptions

1. We start with input parameters Arrays A, B, C, and K.
2. 1-Based Indexing is used in this algorithm.
3. The function SIZE(Arr) returns the number of elements in an array in $O(1)$ time.
4. $1 \leq K \leq |A| + |B| + |C|$ (Ensures the existence of the Kth smallest element).

2 Preprocessing

There is no preprocessing done. The 3 arrays A, B, and C are not modified and are used directly.

3 Algorithm Description

3.1 Broad Overview

The algorithm employs a nested binary search approach. The subroutine used in this algorithm is as follows: It picks an array and assumes the existence of the Kth smallest element in it and then uses binary search to locate it. Each step in the binary search relies on the count of elements less than the pivot element in the union of all arrays ($A \cup B \cup C$).

To find this in the arrays apart from the chosen array, where the count is simply the middle element index chosen during a step in binary search, we use another binary search.

Finally, if the Kth smallest element is found, we return it; otherwise, we repeat the aforementioned steps. It is guaranteed that the Kth smallest element will be found by the time we complete this subroutine in the last array (Assumption 3).

3.2 Detailed Algorithm Description

1. Pick array A and assume that it contains the Kth smallest element.
2. Use Find Binary search on A to locate the Kth smallest element in it. To do this, initialize lo and hi. Then use the while condition while($lo < hi$). Calculate the mid based on the values of hi and lo as:
$$\text{mid} = lo + \frac{lo + hi}{2}$$
3. For every step in Find binary search, assign a pivot, which will be the middle element of the region in which we are performing the binary search.
4. Find the count of elements that are less than or equal to the pivot in B and C and add it to the index of the pivot. This gives us COUNT.
$$\text{pivot} = A[\text{mid}]$$
$$\text{COUNT} = \text{mid} + \text{Counting\_BS}(B, \text{pivot}) + \text{Counting\_BS}(C, \text{pivot})$$

5. To find the count of elements less than or equal to the pivot in an array, we employ Counting Binary Search. Here, initialize $lo = 0$, $hi = \text{SIZE(Arr)}$, and find mid $= lo + \frac{hi-lo}{2}$. Use the while condition while($lo < hi$) and check if mid $\leq$ pivot. If that is the case, store the count as mid and update lo as $lo = \text{mid} + 1$; otherwise, update hi as $hi = \text{mid} - 1$.

6. For Find binary search, if COUNT $= K$, the pivot is the Kth smallest element, and we return it. Else if COUNT $< K$, update lo as $lo = \text{mid} + 1$; else update hi as $hi = \text{mid} - 1$.

7. After the outer binary search ends, we either find the element, which is the Kth smallest element or not.

8. If we do find the Kth smallest element, we return it. Otherwise, we repeat steps 1 to 7 for B and C. In doing so, treat the arrays not chosen the exact same way as we are treating B and C in the above subroutine. Assumption 3 guarantees that we will find the Kth smallest element after performing this subroutine on A, B, and C.

4 Recurrence Relation

This algorithm employs a Divide and Conquer approach. There is no recursion used in this algorithm. This is an iterative algorithm. Therefore, there is no recurrence relation. However, if we implement the same logic recursively, the following will be the recursive relation:

$$T(|A|,|B|,|C|) = \text{Find\_BS}(|A|) + \text{Find\_BS}(|B|) + \text{Find\_BS}(|C|) \quad \text{-(i)}$$

$$\text{Find\_BS}(|A|) = \text{Find\_BS}(|A|/2) + \text{Count\_BS}(|B|) + \text{Count\_BS}(|C|) + O(1) \quad \text{-(ii)}$$

$$\text{Count\_BS}(n) = \text{Count\_BS}(n/2) + O(1) \quad \text{-(iii)}$$

5 Complexity Analysis

Using the above equations, we have the following:

Masters Theorem states that for $T(n) = aT(n/b) + O(n^k \log^p n)$, $T(n) =$

$$\begin{cases} O(n^{\log_b a}) & \text{if } a > b^k \\ O(n^{\log_b a} \log^{p+1} a) & \text{if } a = b^k \text{ and } p > -1 \\ O(n^{\log_b a} \log\log n) & \text{if } a = b^k \text{ and } p = -1 \\ O(n^{\log_b a}) & \text{if } a = b^k \text{ and } p < -1 \\ O(n^k \log^p n) & \text{if } a < b^k \text{ and } p \geq 0 \\ O(n^k) & \text{if } a < b^k \text{ and } p < 0 \end{cases}$$

Using this theorem and (iii), we have $\text{Count\_BS}(n) = O(\log_2 n)$ —(iv).

Using (ii) and (iv), we have $\text{Find\_BS}(|A|) = \text{Find\_BS}(|A|/2) + O(\log_2 |B|) + O(\log_2 |C|) + O(1)$ -(V).

Next, by using the method of substitution, we have

$$\text{Find\_BS}(|A|/2) = \text{Find\_BS}(|A|/4) + O(\log_2 |B|) + O(\log_2 |C|) + O(1)$$
$$\text{Find\_BS}(|A|/4) = \text{Find\_BS}(|A|/8) + O(\log_2 |B|) + O(\log_2 |C|) + O(1),$$

etc.

By substituting all of them into (V) and by utilizing the fact $\text{Find\_BS}(|A|) = O(\log|A|(\log|B| + \log|C|))$ —(VI).

Finally, by using (VI) and (i) we have

$$\begin{aligned} T(|A|,|B|,|C|) = O(&\log|A|(\log|B| + \log|C|) \\ &+ \log|B|(\log|A| + \log|C|) + \log|C|(\log|A| + \log|B|)), \end{aligned}$$

and by setting $|A| = |B| = |C| = n$ as per our assignment statement, we get the final complexity as

$$T(n) = 3 \times O(\log_2^2 n)$$

i.e

$$T(n) = O(\log_2^2 n)$$

6 Pseudocode

Algorithm 1 Your Algorithm

▷ This function Counts the number of elements lower than pivot in the array Arr
function COUNTING_BS($Arr, pivot$)
    $lo \leftarrow 0, hi \leftarrow$ SIZE(Arr)
    $Count \leftarrow 0$
    while ($lo \leq hi$) **do**
        $mid \leftarrow \dfrac{lo + hi}{2}$
        if $Arr[mid] \leq pivot$ **then**
            $Count \leftarrow mid$
            $lo \leftarrow mid + 1$
        else
            $hi \leftarrow mid - 1$
        end if
    end while
    return $Count$
end function
▷ This function finds kth smallest element of A U B U C, assuming it is in A
function FINDING_BS(A, B, C, K)
    $n_A \leftarrow$ SIZE(A)
    $lo \leftarrow 0, hi \leftarrow n_A$
    $Count \leftarrow 0$
    $Res \leftarrow -1$
    $Check \leftarrow 0$
    while ($lo \leq hi$) **do**
        $mid \leftarrow \dfrac{lo + hi}{2}$
        $Pivot \leftarrow A[mid]$
        $Count \leftarrow (mid - lo + 1) +$ Counting_BS(B, pivot) + Counting_BS(C, pivot)
        if ($Count = K$) **then**
            $Check \leftarrow 1$
            $Res \leftarrow A[mid]$
            break
        else if $Count < K$ **then**
            $lo \leftarrow mid + 1$
        else
            $hi \leftarrow mid - 1$
        end if
    end while
    return $Check, Res$
end function
▷ This function finds kth smallest element of A U B U C
function K_THSMALLEST(A, B, C, K)
▷ Assuming Kth smallest element is in array A
    $Check, Res \leftarrow$ Finding_BS(A, B, C, K)
    if ($Check = 1$) **then**
        return Res
    else
▷ Assuming Kth smallest element is in array B
        $Check, Res \leftarrow$ Finding_BS(B, A, C, K)
        if ($Check = 1$) **then**
            return Res
        else
▷ Assuming Kth smallest element is in array C
            $Check, Res \leftarrow$ Finding_BS(C, A, B, K)
            if ($Check = 1$) **then**
                return Res
            end if
        end if
    end if
end function

7 Proof of Correctness

Given:

We are given 3 individual sorted arrays A, B, C with sizes $|A|$, $|B|$, and $|C|$ respectively. The problem states that we need to find the k-th smallest element of the array $(A + B + C)$.

To Prove:

The algorithm proposed in this section solves the given problem efficiently.

Proof:

Since we need to find the k-th smallest element of the array $(A + B + C)$, it is certain that this element will lie in either A, B, or C. Our algorithm assumes this and sequentially tries to exploit each array to find it, using a sequence of 3 binary search-like algorithms per trial (number of trials = 3).

Now, since our output is supposed to be in one of the three arrays, it is acceptable to search in the arrays one by one. This ensures that all possibilities are considered.

Going down to the sequence of 3 binary search-like algorithms on an array: Suppose the sequence is currently experimented on array A. For this trial, we assume that the element lies in array A, whose size is $|A| = n$. Using the first binary search-powered algorithm, we choose an element at a time and count elements less than it in the combined array. If the count is less than $k - 1$, the result lies in the left part of the array. If the count is greater than k, the result lies in the right part of A; otherwise, the chosen element is our result (when the count is equal to $k - 1$).

This can be done only because A is sorted, as given.

Now, if we keep doing this, we might reach a dead end or find the result. We claim that the dead end will be the search space becoming null. By virtue of mathematics, in this finding version of Binary Search, we are experimenting over the index space of the array, i.e., 0 to $n - 1$. After each experiment, the test space becomes half due to the sorted nature of the array, so we have

$$\text{TestRange} = \left\lfloor \frac{n}{2} \right\rfloor$$

For any n and $a \in N$, we have the following:

- For $n = 2a$ (or n is even), new Test Range $= a$
- For $n = (2a) + 1$ (or n is odd), new Test Range $= a$

Thus, the test space boils down to 0 to $a - 1$, which again follows the above-mentioned hypothesis, until a becomes equal to 1 or the test range becomes a single index. If we further try to go on another test level, we end up with a null space or a dead end. So our search either exits earlier with a result or approaches the dead end with no result. Thus, our claim is true, or Binary Search converges.

As for the working of counting binary search, the above-mentioned proof of correctness also works there. Since this algorithm also focuses on taking advantage of a sorted array, we choose an element at a time and check whether it is less than or equal to the pivot. If yes, it is evident that the biggest element of the array which is less than or equal to the pivot (the good element of the array) is either the chosen element or lies in the right side of the array. In the other case, it lies in the left side of the array.

We repeat the binary search steps until we reach the already said good element of the array or our test space becomes null (i.e., there are zero elements less than the pivot). This could again be proved using the above-mentioned hypothesis.

So we have elements less than pivot in A (index[pivot]), elements less than pivot in B (result of countbs(B, pivot)), elements less than pivot in C (result of countbs(C, pivot)). Going back to our array A where we assumed the solution or result to lie, we may actually end up with a solution or we may not.

Now, in case we don't get a result in A, we try to find it in B. Still, if we don't end up with the result, we search in the last array C. And now it's certain that we will end up with a result, which is the solution for this problem. This is true because the solution is supposed to be in either of the three arrays, and we have already shown that our combination of 3 binary search-like algorithms can very well determine if an array has a solution or not.

Thus, after 3 trials of our experimentation, our algorithm is efficiently (in $\mathcal{O}(\log|A|(\log|B| + \log|C|) + \log|B|(\log|A| + \log|C|) + \log|C|(\log|A| + \log|B|))$) able to find the required solution to the problem.