Theory Assignment-1: ADA Winter-2024

Vikranth Udandarao (2022570) Ansh Varshney (2022083)

1 Preprocessing

In this problem, no specific preprocessing steps are required. We assume or have been given the following:

1. A, B, C are sorted arrays.
2. k lies between 1 and 3n.
3. Indices for all arrays start from 0 and end at $n-1$.
4. Comparing two elements can be done in O(1) time.
5. The same numbers are considered unique in different arrays, i.e if the numbers in the union are 1,2,2,3 and we have to find 3rd smallest element, then it will be 2.

2 Algorithm Description

The algorithm uses a nested binary search approach to find the k-th smallest element in the union of three sorted arrays A, B, and C. In the KTH SMALLEST ELEMENT function, we first find the maximum (called high) and minimum (called low) element of the Union Array (say $A' = A \cup B \cup C$) by comparing $A[0]$, $B[0]$, $C[0]$, and $A[n-1]$, $B[n-1]$, $C[n-1]$ respectively.

We use binary search with mid being the middle of the range [*low*, *high*]. Now, the helper function RECURSIVE-BINARYSEARCHCOUNT is called for each array separately.

RECURSIVEBINARYSEARCHCOUNT is a recursive binary function that finds the number of elements that appear before mid in the array. We find this value for each function and then add them to evaluate the variable *totalCount* in KTH SMALLEST ELEMENT function.

If *totalCount* is less than k, then our answer would lie in the half right of mid, so the algorithm shifts *low* to $mid+1$. If *totalCount* is greater than or equal to k, our answer would lie in the half left of mid, so *high* shifts to *mid* (as the answer can be equal to mid also). We return *low* after finishing this search, which is precisely the k-th smallest element in A'.

3 Recurrence Relation

The time complexity of the algorithm can be expressed by the recurrence relation

$$T(n) = T(\log n) + T(\log n)$$

where n is the size of the input arrays A, B, and C. This recurrence relation represents the binary search operations on each array, which leads to the overall $O(\log^2 n)$ time complexity.

The recurrence relation suggests that at each level of the binary search, we are dividing the problem into two subproblems, each of size $\log n$. Since we perform this operation recursively on each subproblem, we have a total of $O(\log^2 n)$ binary search operations.

4 Complexity Analysis

The time complexity of the algorithm is $O(\log^2 n)$, where n is the size of the input arrays A, B, and C. This is achieved through binary search operations on each array and subsequent convergence of the search space. The space complexity is $O(1)$ as the algorithm uses a constant amount of additional space for variables.

The $O(\log^2 n)$ time complexity implies that the algorithm efficiently finds the k-th smallest element with significantly reduced computational effort compared to linear search methods.

5 Pseudocode

Algorithm 1 Kth Smallest Element

```
function recursiveBinarySearchCount(arr, target, low, high)
    if low > high then
        return 0
    end if
    mid ← low + (high − low)/2
    if arr[mid] ≤ target then
        return mid − low + 1+ recursiveBinarySearchCount(arr, target, mid + 1, high)
    else
        return recursiveBinarySearchCount(arr, target, low, mid − 1)
    end if
end function

function KthSmallestElement(A, B, C, k)
    low ← min(A[0], B[0], C[0])
    high ← max(A[n − 1], B[n − 1], C[n − 1])
    while low < high do
        mid ← low + (high − low)/2
        a_count ← recursiveBinarySearchCount(A, mid, 0, size of A − 1)
        b_count ← recursiveBinarySearchCount(B, mid, 0, size of B − 1)
        c_count ← recursiveBinarySearchCount(C, mid, 0, size of C − 1)
        total_count ← a_count + b_count + c_count
        if total_count < k then
            low ← mid + 1
        else
            high ← mid
        end if
    end while
    return low
end function
```

6 Proof of Correctness

Let's assume that the arrays A, B, and C are sorted in ascending order. Also, let A' be the union of A, B, and C. We want to find the k-th smallest element in A'.

The algorithm uses a binary search approach to efficiently narrow down the search space. The variable *low* represents the minimum possible value for the k-th smallest element, and *high* represents the maximum possible value.

In each iteration of the while loop, the algorithm calculates the middle value *mid* between *low* and *high* and

counts the number of elements less than or equal to mid in each of the arrays A, B, and C using the RECURSIVEBINARYSEARCHCOUNT function.

The total count is then calculated as the sum of counts from all three arrays. If the total count is less than k, it means that the k-th smallest element must be in the right half of the current search space. Therefore, the algorithm updates low to $mid + 1$. If the total count is greater than or equal to k, the k-th smallest element must be in the left half of the current search space, and $high$ is updated to mid.

The algorithm continues this process until low and $high$ converge, at which point low represents the k-th smallest element.

Now, let's consider the correctness of the algorithm:

1. Initialization: - At the beginning of the algorithm, low is set to the minimum element in A', and $high$ is set to the maximum element in A'. This ensures that the search space is initialized correctly.

2. Maintenance: - In each iteration of the while loop, the search space is narrowed down by updating either low or $high$ based on the total count of elements less than or equal to mid. The algorithm correctly maintains the search space to eventually converge to the k-th smallest element.

3. Termination: - The while loop terminates when low and $high$ converge. At this point, low represents the k-th smallest element in A'. The algorithm correctly identifies the k-th smallest element within the sorted union of arrays.

Therefore, the algorithm is correct and efficiently finds the k-th smallest element in the union of three sorted arrays.