

Theory Assignment-1: ADA Winter-2024

Vikranth Udandaraao (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 `RECURSIVEBINARYSEARCHCOUNT` 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 algorithm used is not recursive in nature and so, there will not be any recurrence relation for the whole algorithm. However, in the implementation of the `RECURSIVEBINARYSEARCHCOUNT` function, recursion has been used. So, the recurrence relation (**ONLY FOR** `RECURSIVEBINARYSEARCHCOUNT` function) will be:

$$T(n) = T(n/2) + c$$

which will solve to get the complexity as $O(\log(n))$ (**ONLY FOR** `RECURSIVEBINARYSEARCHCOUNT` function).

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 . We can derive this as follows

1. The first binary search operates in the range of $3n$ elements, covering all elements of the arrays.
2. For each iteration in the binary search, three different binary searches are performed on each array, each with n elements.
3. As each binary search on m elements has a time complexity of $O(\log m)$, the overall complexity becomes $O(\log(3n) \cdot 3 \log(n))$, simplifying to $O(\log^2 n)$

The $O(\log^2 n)$ time complexity signifies the algorithm's efficiency in finding the k -th smallest element with significantly reduced computational effort compared to linear search methods.

5 Pseudocode

Algorithm 1 Kth Smallest Element

```
1: function RECURSIVEBINARYSEARCHCOUNT(arr, target, low, high)
2:   if low > high then
3:     return 0
4:   end if
5:   mid  $\leftarrow$  low + (high - low)/2
6:   if arr[mid]  $\leq$  target then
7:     return mid - low + 1 + RECURSIVEBINARYSEARCHCOUNT(arr, target, mid + 1, high)
8:   else
9:     return RECURSIVEBINARYSEARCHCOUNT(arr, target, low, mid - 1)
10:  end if
11: end function

12: function KTHSMALLESTELEMENT(A, B, C, k, 3n)
13:   low  $\leftarrow$  min(A[0], B[0], C[0])
14:   high  $\leftarrow$  max(A[n - 1], B[n - 1], C[n - 1])
15:  //Running the first binary search.
16:  while low < high do
17:    mid  $\leftarrow$  low + (high - low)/2
18:    a_count  $\leftarrow$  RECURSIVEBINARYSEARCHCOUNT(A, mid, 0, n - 1)
19:    b_count  $\leftarrow$  RECURSIVEBINARYSEARCHCOUNT(B, mid, 0, n - 1)
20:    c_count  $\leftarrow$  RECURSIVEBINARYSEARCHCOUNT(C, mid, 0, n - 1)
21:    total_count  $\leftarrow$  a_count + b_count + c_count
22:  //totalCount stores the number of elements before mid in A'.
23:    if total_count < k then
24:      low  $\leftarrow$  mid + 1
25:    else
26:      high  $\leftarrow$  mid
27:    end if
28:  end while
29:  return low
30: end function
```

6 Proof of Correctness

6.1 Dry Run

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.

DEMO TESTCASE: $A=[2,3,6]$ $B=[1,5,8]$ $C=[4,7,9]$; $n=3$; $k=2$

OUTPUT: 2

EXPLANATION: The first binary search runs in the range $[1,9]$. We keep on finding the values for `totalCount` till we reach the desired k value. We will return *low* which will be equal to 2.

6.2 Induction

Let's use induction to prove the correctness of the algorithm:

Base Case ($k = 1$): For the base case where $k = 1$, the algorithm aims to find the smallest element in the union of arrays A , B , and C . The algorithm initializes *low* to the minimum element in A' , and *high* to the maximum element in A' . In this case, *low* and *high* will converge to the smallest element in A' , which is the correct result. Therefore, the base case holds true.

Inductive Step: Assume that the algorithm correctly finds the k -th smallest element for some arbitrary k (inductive hypothesis). We want to show that it also correctly finds the $(k + 1)$ -th smallest element.

Consider the $(k + 1)$ -th iteration of the algorithm, where *low* and *high* are updated based on the total count of elements less than or equal to *mid*. If the total count is less than $(k + 1)$, the $(k + 1)$ -th smallest element must be in the right half of the current search space (*low* is updated to $mid + 1$). If the total count is greater than or equal to $(k + 1)$, the $(k + 1)$ -th smallest element must be in the left half of the current search space (*high* is updated to *mid*).

Now, we know that the inductive hypothesis holds for k . Therefore, the k -th smallest element is correctly identified in the union of arrays. The $(k + 1)$ -th iteration ensures that the $(k + 1)$ -th smallest element is correctly identified in the updated search space.

Since the algorithm correctly identifies the 1st smallest element in the base case and preserves correctness in the inductive step, by induction, we can conclude that the algorithm correctly finds the k -th smallest element for any valid k in the union of three sorted arrays.

This completes the inductive proof of correctness for the given algorithm.