Theory Assignment-1: ADA Winter-2024

Himang Chandra Garg (2022214) Nishil Agarwal (2022334)

1 Preprocessing

Not Applicable in this Algorithm.

2 Algorithm Description

We have used a modified binary search algorithm to solve the problem.

First, we find the largest and the smallest of all three arrays altogether by comparing the initial and final values of the arrays. We then use the 'largest' and 'smallest' values to find an average value of them and call this value an "assumed middle" value. After that, we count the total number of elements in each array that are smaller than this middle value.

We aim to bring the "assumed middle" value nearest to the kth smallest element. We achieve this by trying to get the 'smallest' and 'largest' values closer to kth smallest element through recursion. As a result, their average i.e. "assumed middle" gets closer to the kth value. We monitor the closeness of these values to kth smallest number by ensuring that, the total number of values in all arrays altogether that are lesser than the "assumed value" is nearing to k.

The modified binary search for this works in the following manner :

We divide the array size in half and check if the "assumed middle" value lies to the left or right half of the array (We stop immediately if it lies right in the middle). Recursively, we divide the array into half again and choose the half containing values near to "assumed middle." We continue this until the size of the array becomes one, indicating that we have reached the nearest value to the "assumed value," which we call, let's say - GUESS.

Further algorithm :

1)If the 'smallest' becomes equal to 'largest' we stop. When they become equal it essentially means that we have reached the kth smallest value.

2)If the total is smaller than k, then our "assumed middle" is definitely smaller than the kth smallest element. So, we replace the 'smallest' value used in calculation of the "assumed middle" with the current "assumed middle" and recalculate it. Thus, we can get a higher "assumed middle" value which is possibly closer to the kth smallest element.

3)If the total is greater than k, then our "assumed middle" is definitely larger than the kth smallest element. So, we replace the 'largest' value used in calculation of the "assumed middle" with the current "assumed middle" and recalculate it. Thus, we can get a lower "assumed middle" value which is possibly closer to kth smallest element.

We repeat this process till 'smallest' becomes equal to 'largest'. At that instant, the smallest'/'largest' is basically the kth smallest value which we return as our answer.

3 Recurrence Relation

$$T(R) = T(R/2) + O(\log(n)) + c$$

We came to the above recurrence relation by analyzing our recursive search function in which, in each function call, we find the "assumed middle" value and compare the number of elements smaller than that "assumed middle" value. According to this comparison we call the same function recursively after updating the value of 'smallest' or 'largest' to 'assumed middle' in the next iteration. This makes our 'assumed middle' half every time we call the function. So, it takes T(R/2) time in total. Here, R is the initial 'assumed middle'.

Here the O(log(n)) part comes from the recursive modified binary search function in which, in each function call, we search to find an index of an element within the list, to the left of which all elements are lesser than the required value. With each recursive call, the size of the list reduces to half, hence requiring half the time.

Here, c is the constant time taken for other comparisons and calculations.

4 Complexity Analysis

The time complexity of this Algorithm is

$$O(\log(n)*\log(R))$$

Here, n is the size of array.
R is the range that is the Maximum element - Minimum element.

We came to this result since, in our recursive function, we are calling a modified binary search on each list, which itself has a time complexity of O(log(n)). This is because with each iteration of the search the list elements are halved. Also, the recursive function responsible for setting the "Assumed middle" value itself is called log(R) times in the worst case. This is because the difference is reduced to half after each function call. The kth smallest number must lie somewhere in between the largest and smallest number in the union.

Our code only has a few temporary variables and no extra space has been used for storing anything. Thus, the space complexity of our algorithm is O(1).

5 Pseudocode

Auxiliary Functions

```
1: function LARGEST3(num1, num2, num3)          ▷ Returns largest of given 3 numbers
2:     return max(max(num1, num2), num3)
3: end function
4: function SMALLEST3(num1, num2, num3)         ▷ Returns smallest of given 3 numbers
5:     return min(min(num1, num2), num3)
6: end function
```

Algorithm 1 Modified Binary Search

```
function MODIF_BINS(arr, value, low, high)
    mid ← (low + high)/2
    if low > high then
        if low = 0 then
            return mid
        else
            return mid + 1
        end if
    else if mid = 0 then
        if arr[mid] > value then
            return 0                                   ▷ No element smaller than value in array
        else
            return 1                                   ▷ The smallest element in array is smaller than value
        end if
    else if arr[mid] ≥ value and arr[mid − 1] ≤ value then
        if arr[mid] = value then
            return mid + 1                             ▷ All elements till index 'mid' are smaller
        else
            return mid                                 ▷ All elements till index 'mid-1' are smaller
        end if
    else if arr[mid] < value and arr[mid − 1] < value then
        return MODIF_BINS(arr, value, mid + 1, high)
    else if arr[mid] > value and arr[mid − 1] > value then
        return MODIF_BINS(arr, value, low, mid − 1)
    end if
end function
```

Recursive "Assumed middle" kth value nearing function

```
function RECUR_SEARCH(k, n, arr1, arr2, arr3, smallest, largest)
    ▷ Initially, smallest and largest are passed by auxiliary function, and initial and final values of each list
    assumed_middle ← (largest + smallest)/2
    if (smallest + largest) < 0 then
        assumed_middle ← (largest + smallest − 1)/2
    end if
    r1 ← MODIF_BINS(arr1, 0, n − 1, assumed_middle)
                                                       ▷ Count of elements smaller than assumed_middle in array 1
    r2 ← MODIF_BINS(arr2, 0, n − 1, assumed_middle)
    r3 ← MODIF_BINS(arr3, 0, n − 1, assumed_middle)
    total ← r1 + r2 + r3
                        ▷ Total number of elements in all arrays altogether which are lesser than assumed_middle
    if smallest = largest then
        return smallest
                                                       ▷ kth smallest value found
    else if total > k then
        return RECUR_SEARCH(k, n, arr1, arr2, arr3, assumed_middle + 1, largest)
                                                       ▷ Assumed value is still larger than kth smallest
    else if total < k then
        return RECUR_SEARCH(k, n, arr1, arr2, arr3, smallest, assumed_middle)
                                                       ▷ Assumed value is still smaller than kth smallest
    end if
end function
```

6 Proof of Correctness

In this question we calculate a value called “Assumed middle” and continuously, recursively try to take it closer to kth smallest number. This is very accurate as our modified binary search checks at each step if our “Assumed middle” value is getting closer to kth smallest number or getting away from it. We check this by counting a total number of elements lesser than the “assumed middle” in all the arrays. Our approach is to make this total number of elements equal to k to get the kth smallest number. We also terminate further recursions if the smaller bound of the ”assumed middle” becomes equal to larger bound. (i.e., the solution has been found already)

Our c++ Code (already tested with many test cases) is:

`https://drive.google.com/file/d/1yEaf5MxFe7N6iTk39s7CDRii3F3ScC3P/view?usp=sharing`