Theory Assignment-1: ADA Winter-2024

Himanshu Raj (2022216) Tanish Verma (2022532)

January 27, 2024

1 Preprocessing

Not Applicable

2 Algorithm Description

The FindK function specifies its 'low' as the minimum of all elements in the three arrays and 'high' as the maximum of all elements. It finds a middle value between high and low and checks how many elements are less than or equal to the 'mid' value in all arrays. If the count is more than equal to k, it means the kth element must lie on the left of the middle value and set high to mid-1; and if the count is less than k, it means the kth element must lie on the right of the middle value and sets low as mid+1, thus effectively reducing problem size by 2 (divide and conquer).
This utilizes the fact that the arrays are sorted to ensure the left half has elements smaller than mid and the right has elements greater than mid.

We are using BinaryCount as a helper function to count less than equal to elements from the mid value. It simply uses binary search over the entire array for the element that is less than equal to the key(mid) and returns its index, which is essentially the count of the elements which are less than equal to key(mid).

*Indexing is one-based (1,2,...,n)
*Constraints over k: 1<=k<=3n

3 Recurrence Relation

Even though it isn't asked for the iterative approach, we're still mentioning it here.

$$T(n) = T(n/2) + 3(log_2(n)), \quad T(1) = 1$$

After solving the recurrence relation, we get

$$T(n) = O(log^2(n))$$

4 Complexity Analysis

For an iterative binary search algorithm,
the first comparison looks at $n/2^{th}$ element, second comparison at $n/4^{th}$ and so on. So the k^{th} comparison at $n/2^{kth}$ position. So, a tree is formed of depth log(n). This is because the base case for this tree can be $n/2^k = 1$ which implies k = $\log_2$(n). So, k is

$$O(log_2(n))$$

Now, the FindK function is another binary search implementation, and the number of elements which it works upon is len = max(A[n], B[n], C[n]) - min(A[1], B[1], C[1]) This function involves O(1) steps, but includes a step in which it calls the BinaryCount function three times. For that step, the time complexity is

$$O((log_2(n) + log_2(n) + log_2(n))$$

which is just

$$O(log_2(n))$$

Also, the FindK function itself is a binary search of "len" number of elements, and at each step takes O($\log_2$(n)) time. Therefore overall time complexity is

$$O((log_2(n) * (log_2(max(A[n], B[n], C[n]) - min(A[1], B[1], C[1])))$$

which is equivalent to,

$$O(log^2(n))$$

The worst case and average case complexities are O($\log^2(n)$).

5 Pseudocode

Algorithm 1 Your Algorithm

```
function FindK (A, B, C, n, k)
    low = min(A[1], B[1], C[1])
    high = max(A[n], B[n], C[n])
    while (low <= high):
        mid = low + (high-low)/2
        count = BinaryCount(A, n, mid) + BinaryCount(B, n, mid) + BinaryCount(C, n, mid)
        if (count >= k):
            high = mid - 1
        else:
            low = mid + 1
    return low
end function

function BinaryCount (arr, len, key)
    low = 1
    high = len
    while (low <= high):
        mid = low + (high-low)/2
        if (arr[mid] <= key):
            low = mid + 1
        else:
            high = mid - 1
    return low
end function
```

6 Proof of Correctness

1. Since there are constraints over k, i.e. 1<=k<=3n, a k^{th} element must exist in the union of 3 arrays.
2. A sorted array has the property that A[i]th element is greater than or equal to i-1 elements of that array which occur before it.

The high and low are the maximum and minimum of all elements in the union of three arrays respectively, so the kth element must be an element present in the array which is between high and low.
The algorithm we have implemented uses these properties to find the position of an element.
The BinaryCount function simply returns the number of elements which are smaller than the key provided, and the sum of all these counts (of the three arrays) provides the number of elements which are lesser than the current mid, thereby providing the index of the element essentially (refer property 2). If the count is more than equal to k, it means the kth element must lie on the left of the middle value and set high to mid-1; and if the count is less than k, it means the kth element must lie on the right of the middle value and sets low as mid+1, thus effectively reducing problem size by 2 (divide and conquer).

Since the existence of such a k (which satisfies the problem solution) always follows (refer property 1), we can find such a k through a binary search on the basis of the count returned for that key.