Theory Assignment-1: ADA Winter-2024

Devansh Grover (2022151) Garvit Kochar (2022185)

Preprocessing:

Given three sorted arrays A, B, and C each having n numbers, we assume that we can compare two elements from $A \cup B \cup C$ in O(1)-time. The algorithm uses the binary search to find the Kth smallest element in $A \cup B \cup C$. To do this, we use the following binary search algorithm:

Algorithm: binarySearch
function binarySearch(Arr, start, end, key)

```
1. if start > end then
2.     return start

3. mid = (start / 2) + (end / 2) + ((start % 2 + end % 2) / 2)

4. If Arr[mid] >=key then
5.     return binarySearch(Arr, mid + 1, end, key)
6. else
7.     return binarySearch(Arr, start, mid - 1, key);
```

end function

Explanation of inputs:
Arr: input array
start: index of starting element
end: index of ending element
key: the value against which elements of the array are compared

The recurrence relation for this algorithm is:
T(n) = T(n/2) + C*(n^0)

The algorithm recursively calls itself with a reduced input size of n/2 in each recursive call, corresponding to dividing the array.

Using Master's Theorem, a=1, b=2 and f(n) = c*(n^0).
Since c*(n^0) is a constant, the Master's Theorem gives the time complexity of the binarySearch algorithm as O(logn).

Algorithm Description

The algorithm is implementing a variant of the binary search algorithm to find the 'K-th' smallest element in $A \cup B \cup C$. This is because the arrays A, B and C are given to us as sorted arrays

which is a prime opportunity to use the binary search. The function findKth is the main function which calculates the Kth smallest element. It uses the helper function binarySearch to find the position of a given element in the array. **Please note that we use 0 based indexing here**, therefore, if there are n elements in an array then the indexing will be from 0 to n-1.

binarySearch

The binarySearch function is used to search for a given key in an array between the indices start and finish. The index of the first element greater than the key is returned. It yields end + 1 if all elements are less than or equal to the key.

findKth

- The findKth function checks for base cases where k is out of bounds or the search range (start to end) is invalid. If k is less than or equal to 0 or greater than n*3, the function returns -1 (indicating an invalid case).
- Inside the main recursive logic, the function calculates the middle index (mid) in a similar manner as in the binarySearch function.
- It then uses the binarySearch function to find the count of elements less than or equal to mid in each of the three arrays (A, B, and C).
- If the total count of such elements is less than k, the kth smallest element must be in the right half of the current range. Therefore, the function recursively calls itself with an updated search range (mid + 1 to end).
- If the total count is greater than or equal to k, the kth smallest element must be in the left half of the current range. The function recursively calls itself with an updated search range (start to mid).
- The recursion continues until the search range is reduced to a single element (start == end), at which point the function returns the position of the kth smallest element.

Recurrence Relation

The algorithm recursively calls itself with a reduced input size of n/2, corresponding to the division of the input arrays in each recursive call. The binarySearch function has a time complexity of O(logn), as mentioned above. It is called on the 3 arrays separately within each recursive call, i.e, 3*logn. The binary search is performed on each of the three arrays (A, B, C), and the sum of the counts from the three binary searches contributes to the O(log n) factor.

Therefore, the recurrence relation for the algorithm is of the form:
$T(n) = T(n/2) + 3*\log n$

It can be alternatively written as $T(n) = T(n/2) + c*\log n$, for c=3.

Complexity Analysis

Using the recurrence relation

$T(n) = T(n/2) + 3*\log n$

Construct the recursive tree
Start with the initial term T(n), and at each level, break the problem down into subproblems until you reach the base case.

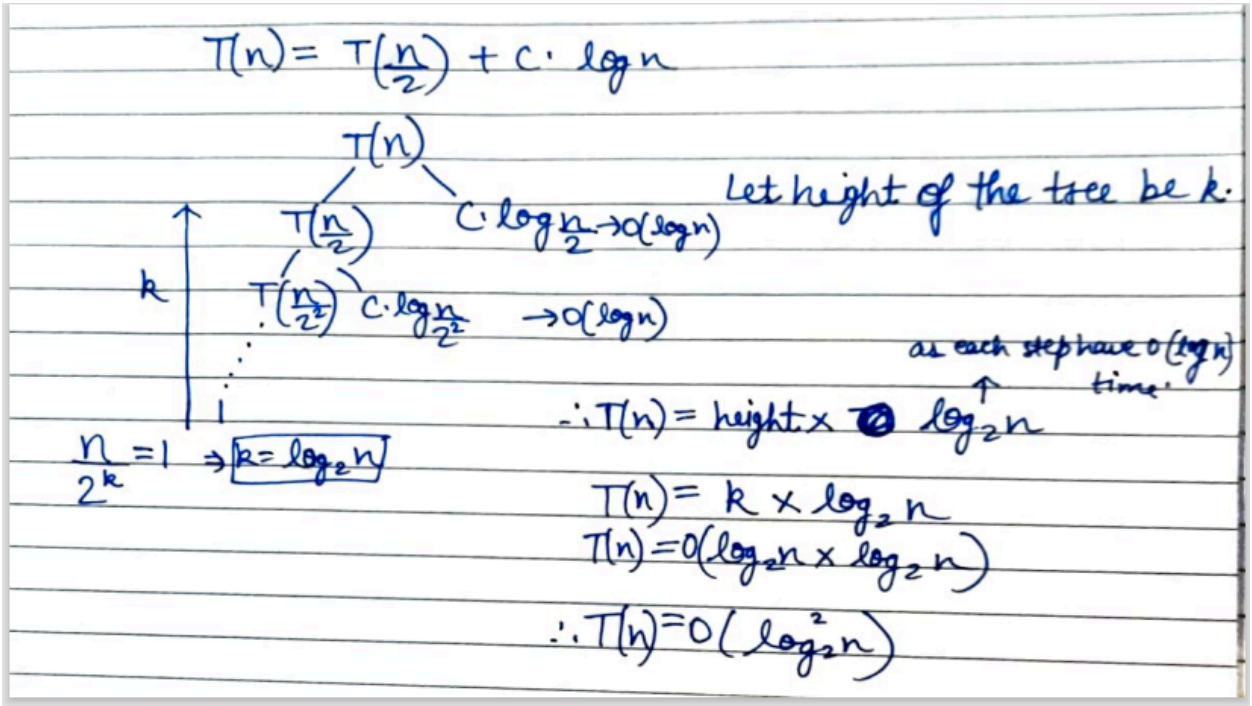
```
                T(n)
                 |
T(n/2)          3*(logn/2)
                 |
T(n/4)          3*log(n/4)
                 |
 T(1)            O(logn)
```

Analyze the cost at each level:
At each level of the tree, we have a cost of O(logn). Since we are dividing the problem size by 2 at each step, there are logn levels in the tree.

Sum up the costs:
Sum up the costs at each level to get the total cost of the algorithm.
T(n) = O(logn) + O(logn) + … + O(logn)
where there are logn terms
T(n) = O(logn*logn) = O((logn)^2)

Hence, the time complexity of the algorithm is O((logn)^2).

The attached hand-written picture is given to better explain the time complexity:



Pseudocode

Algorithm: findKth

function findKth(A, B, C, n, k, start, end)

```
1. if k <= 0 or k > n*3 then
2.     return -1

3. if start < end then
4.     mid = (start / 2) + (end / 2) + ((start % 2 + end % 2) / 2)
5.     LesserInA = binarySearch(A, 0, n-1, mid)
6.     LesserInB = binarySearch(B, 0, n-1, mid)
7.     LesserInC = binarySearch(C, 0, n-1, mid)

8.     if LesserInA + LesserInB + LesserInC < k then
9.         return findKth(A, B, C, n, k, mid + 1, end)
10.    else
11.        return findKth(A, B, C, n, k, start, mid)

12. else
13.    return start
```

end function

Explanation of inputs:
A, B, C: the input arrays
start: INT_MIN
end: INT_MAX
n: number of elements in each array
k: the k-th smallest element to be found

Proof of Correctness:

The binarySearch function operates by establishing a base case where it returns the start index when start surpasses end, signifying the appropriate insertion position for the given key in the sorted array. In the recursive case, the function calculates the middle index mid and dynamically adjusts the search range based on the comparison of Arr[mid] with the key. If `Arr[mid] <= key`, it recursively searches in the right half, otherwise in the left half, maintaining the invariant that the correct insertion position for the `key` lies within the current search range [start, end].

The findKth function utilizes the binarySearch function to count the number of elements less than or equal to mid in each of the three arrays (A, B, C). The base case checks if the provided value of k is out of bounds, returning -1 if true. In the recursive case, the function calculates mid and adjusts the search range based on the total count of elements less than or equal to mid in the three arrays. The invariant upheld by the findKth function is that the k-th smallest element in the union of the three arrays (A, B, C) is within the current search range [start, end].