Theory Assignment-1: ADA Winter-2024

Ritika Thakur (2022408) Sidhartha Garg (2022499)

1 Preprocessing

No preprocessing steps required since the three arrays are already sorted.

2 Algorithm Description

The goal is to achieve an algorithm that outputs the k-th smallest element of $A \cup B \cup C$, given that A, B, and C are three sorted arrays, with a time complexity less than $\mathcal{O}(n)$. The algorithm uses binary search and counts the number of elements less than or equal to a given key(mid value) in each array to determine the k-th smallest element.
In the special case where $k < \log n . \log m$ where n is the size of array and m is the range of values in the arrays, performing a linear search for k is more efficient than a binary search on n.

2.1 Iterative Approach (`findKth`)

Initialization:

- Determine the minimum and maximum possible values in the merged array (low and high).
- Create a while loop that continues until low exceeds high.

Midpoint Calculation:

- Calculate the middle value (`mid`) between low and high.

Counting Elements:

- Use `countNums` to count the elements less than or equal to `mid` in each of the three arrays, storing the total count in `cnt`.

Search Space Adjustment:

- If `cnt` is less than `k`, the `k`-th element must lie in the right half of the search space. Update low to `mid + 1`.
- Otherwise, the `k`-th element is in the left half or is `mid` itself. Update high to `mid - 1`.

Final Value:

- When the loop terminates, low points to the `k`-th element in the merged array.

2.2 Recursive Approach (`findKthRecursive`)

Base Case:

- If `low` is greater than `high`, return `low` as the `k`-th element.

Midpoint Calculation:

- Calculate the middle value (`mid`) between `low` and `high`.

Counting Elements:

- Use `countLess` to count the elements less than or equal to `mid` in each of the three arrays, storing the total count in `cnt`.

Recursive Calls:

- If `cnt` is less than `k`, the `k`-th element must lie in the right half of the search space. Recursively call `findKthRecursiveEle` with the updated range (`mid + 1, high`) and adjusted `k` value.
- Otherwise, the `k`-th element is in the left half or is `mid` itself. Recursively call `findKthRecursiveEle` with the range (`low, mid - 1`).

3 Recurrence Relation

The recurrence relation is given by:

$$T(n, m) = T\left(\frac{m}{2}\right) + 3\log_2(n)$$

Let $m = 2^k$, where k is a non-negative integer:

$$T(n, 2^k) = T\left(2^{k-1}\right) + 3\log_2(n)$$

Now, let $A(k) = T(2^k)$. Substituting this into the equation:

$$A(k) = A(k-1) + 3\log_2(n)$$

This is a linear recurrence relation, and its solution is given by:

$$A(k) = A(0) + k \cdot 3\log_2(n)$$

Now, substitute back $k = \log_2(m)$:

$$A(\log_2(m)) = A(0) + 3\log_2(n) \cdot \log_2(m)$$

Finally, substitute back $A(k) = T(2^k)$ to get the solution in terms of $T(n, m)$:

$$T(n, m) = T(1) + 3\log_2(n) \cdot \log_2(m)$$

4 Complexity Analysis

For both iterative and recursive solutions:

4.1 Time Complexity

The time complexity is given by $\mathcal{O}(\log n \cdot \log m)$, which can be approximated as $\mathcal{O}(\log^2 n)$ for simplicity, where n is the size of the input arrays, and m is the range of values in the arrays.
The `CountLess` function employs binary search over arrays of size n, resulting in a time complexity of $\mathcal{O}(\log n)$. The main loop runs for a maximum of $\log m$ iterations (or calls the FindKthEleRecursive function a maximum of $\log m$ times in the recursive approach) due to the halving of the search range. Within each iteration (or recursive call), it calls the `CountLess` function. Hence, the total time complexity becomes $\mathcal{O}(\log n \cdot \log m)$, which can be roughly expressed as $\mathcal{O}(\log^2 n)$.

4.2 Space Complexity

The space complexity is $\mathcal{O}(1)$ for the iterative approach, as no additional space is utilized. For the recursive approach, there is the existence of an extra recursion stack space.

5 Pseudocode

Algorithm 1 Find the kth element among three sorted arrays (Iterative Approach)

```
function FindKthEle(arr1[], arr2[], arr3[], n, k)
    low ← min(min(arr1[0], arr2[0]), arr3[0])
    high ← max(max(arr1[n − 1], arr2[n − 1]), arr3[n − 1])
    while low ≤ high do
        mid ← low + ⌊(high−low)/2⌋
        cnt ← CountLess(arr1, n, mid) + CountLess(arr2, n, mid) + CountLess(arr3, n, mid)
        if cnt < k then
            low ← mid + 1
        else
            high ← mid − 1
        end if
    end while
    return low
end function
function CountLess(arr[], n, key)
    low ← 0
    high ← n − 1
    while low ≤ high do
        mid ← low + ⌊(high−low)/2⌋
        if arr[mid] ≤ key then
            low ← mid + 1
        else
            high ← mid − 1
        end if
    end while
    return low
end function
```

Algorithm 2 Find the kth element among three sorted arrays (Recursive Approach)

```
function FindKthEleRecursive(arr1[], arr2[], arr3[], n, k, low, high)
    if low > high then
        return low
    end if
    mid ← low + ⌊(high−low)/2⌋
    cnt ← CountLess(arr1, n, mid) + CountLess(arr2, n, mid) + CountLess(arr3, n, mid)
    if cnt < k then
        return FindKthEleRecursive(arr1, arr2, arr3, n, k, mid + 1, high)
    else
        return FindKthEleRecursive(arr1, arr2, arr3, n, k, low, mid − 1)
    end if
end function
```

6 Proof of Correctness

6.1 Iterative Approach (`FindKthEle`)

Initialization: The algorithm correctly initializes `low` and `high` to encompass the entire possible range of values in the merged array.

Why it works: At each iteration, the algorithm adjusts the search range based on the count of elements less than or equal to mid in the combined three arrays. If `cnt` is less than `k`, it means there are fewer than `k` elements in the combined three arrays that are less than or equal to `mid`. Therefore, the `k`-th smallest element must be to the right of `mid`. To find it, the algorithm updates `low` to `mid + 1`. If `cnt` is greater than or equal to `k`, it means there are `k` or more elements in the combined three arrays that are less than or equal to `mid`. Therefore, the `k`-th smallest element must be to the left of or equal to `mid`. To find it, the algorithm updates `high` to `mid - 1`.

Termination: The loop terminates when `low` exceeds `high`, indicating the search space has been narrowed to a single element, which is the `k`-th element.

6.2 Recursive Approach (`FindKthEleRecursive`)

Base Case: Correctly handles the scenario when the search space is empty, indicating the `k`-th element has been found.

Recursive Case: Recursive calls correctly divide the problem into smaller subproblems, maintaining the invariant that the `k`-th element lies within the current search range. The recursive calls either find the `k`-th element directly or create subproblems that eventually lead to its discovery.

Key Points: Both algorithms ensure that the `k`-th element is always within the considered search space. They effectively reduce the search space by half in each iteration/recursive call, converging on the correct element. The correctness relies on the assumption that the input arrays are sorted individually.

Additional Notes: The `countNums` function, used within both approaches, is essential for correctly counting elements less than or equal to a given value in a sorted array. Its correctness is based on the binary search algorithm. The time complexity analysis ($\mathcal{O}(\log N \cdot \log M)$) proves that both algorithms are efficient in finding the `k`-th element without fully merging the arrays.