Theory Assignment-1: ADA Winter-2024

Nikhil Kumar (2022322) Nikhil (2022321)

28 January 2024

1 Preprocessing

P̄reprocessing is not needed.

2 Algorithm Description

The Algorithm we used is based on the Binary Search Algorithm. It uses the idea of search spaces efficiently and the fact that three arrays are already sorted. It has mainly two functions - Counter and Find_the_kth_element.

The Counter function performs the function of finding the number of elements less than or equal to a given target and it uses the idea of binary search to do so.

- Three variables are taken as input, with low being 0 and high denoting the number of elements in the array and input for a target.
- The function is called recursively to repeatedly calculate a middle value and narrow down the search space. The condition: low $\leq$ high ensures that we are inside the array. Now we calculate a middle element through: low + $\frac{\text{high}-\text{low}}{2}$, which simply finds the average of low and high and assigns this value to mid. Depending on the value of that middle element, we recursively call this function with updated values of low and high.
- Finally in the base case,when the condition low $\leq$ high fails, we return the value of low as the count of elements less than or equal to the given target.

The Find_the_kth_element function also uses the idea of the binary search.

- Two variables are given as input, with low being the minimum value of the three arrays and high being the maximum of the three arrays. As the three arrays are sorted, this reduces the time for finding the maximum and minimum elements.
- We recursively call this function, which calculates the middle value of the current search space.
- Now in each recursive call, we count the number of elements $\leq$ mid from the three arrays with the help of Counter function. The total count is then calculated by adding the individual sums together. Depending on the value of the count we recursively call our function accordingly with updated values of low and high.
- In the base case of recursion when the condition low $\leq$ high fails we return our answer as low.

3 Recurrence Relation

The recurrence relation can be expressed as follows:

$$T(n) = T\left(\frac{n}{2}\right) + O(\log n)$$

Here,

$T(n)$ is the time complexity of the function for an input of size n.

$T\left(\frac{n}{2}\right)$ represents the recursive calls on a smaller range, as the binary search halves the range in each recursive call.

$O(\log n)$ is the complexity for searching the element smaller than the target using binary search.

4 Complexity Analysis

Time Complexity Analysis:

The main component that describes the time complexity of the code is Binary search operation both in the `Counter` function and the `Find_the_kth_element` function. We have given that each array has a fixed number of elements i.e. n.

1. **Binary Search Operation:**
   - Time Complexity of the Binary search operation in the worst case is $O(\log n)$.
   - In the `Find_the_kth_element` function, For each array, binary search is performed. I.e. Three binary searches are performed in this function.
   - Therefore, the time complexity of a single recursive call is $O(3 \log n)$.
2. **Number of Recursive Calls:**
   - The recursion continues until the range $[low, high]$ till the condition $(low > high)$.
   - In the worst case, the recursion depth is $\log n$, where n is the size of the range or the array.
3. **Overall Time Complexity:**
   - Considering the recursive calls and the binary search operations, the overall time complexity is $O(\log^2 n \cdot \log_2 n)$.

Space Complexity :- The space complexity is primarily described by the function call stack.

1. **Function Call Stack:**
   - The function call stack needs to call the recursion depth.
   - In the worst case, the recursion depth is $\log n$.
   - Therefore, the space complexity due to the call stack is $O(\log n)$.
2. **Additional Space:**
   - The additional space used in each function call is constant (e.g., variables like low, high, mid, and the counts).
   - Hence, the additional space complexity is $O(1)$.
3. **Overall Space Complexity:**
   - The overall space complexity is $O(\log n)$ due to the function call stack.

5 Pseudocode

Algorithm 1 Algorithm to find the kth-smallest element

```
function COUNTER(arr, target, low, high)
    if low > high then
        return low                                  ▷ Returns the count of elements ≤ target
    end if
    mid ← low + (high−low)/2
    if arr[mid] ≤ target then
        return COUNTER(arr, target, mid + 1, high)
    else
        return COUNTER(arr, target, low, mid - 1)
    end if
end function


function FIND_THE_KTH_ELEMENT(A, B, C, k, low, high)
    if low > high then
        return low
    end if
    mid ← low + (high−low)/2
    count ← Counter(A, mid, 0, size(A) - 1) + Counter(B, mid, 0, size(B) - 1) +
Counter(C, mid, 0, size(C) - 1)
    if count < k then
        return FIND_THE_KTH_ELEMENT(A, B, C, k, mid + 1, high)
    else
        return FIND_THE_KTH_ELEMENT(A, B, C, k, low, mid - 1)
    end if
end function
```

6 Proof of Correctness

There are two functions to find the K-th smallest element from the given three arrays.

6.1 Counter Function

1. **Counter**: Returns the count of elements smaller than the target in a sorted array.

   Proof of Counter function correctness:

   (a) This is a standard binary search pattern used to find the count of elements less than or equal to the target in the sorted array.

   (b) The base case (low > high) ensures that the function returns the correct count when the search space is empty.

   (c) The recursive call for this function is made on a reduced search space, either to the left or right of the current midpoint. This also ensures that the array is divided into halves at each step.

   (d) When the base case is reached, the function returns the correct count of elements that are less than or equal to the target.

6.2 Find_the_kth_element Function

1. **Find_the_kth_element**: The variables 'low' and 'high' in the binary search maintain a valid range within which the k-th smallest element can be found in the sorted array.

   Proof of Correctness of the Find_the_kth_element:

(a) In each recursive step, the function calculates the counts of elements less than or equal to the mid-point in each of the arrays A, B, and C using `Counter`.

(b) The function follows a binary search approach to find the k-th smallest element in the sorted arrays A, B, and C.

(c) The function then recursively searches in the appropriate half of the search space.

(d) The correctness of `Counter` ensures that the counts are accurate, and the binary search in `Find_the_kth_element` correctly identifies the location of the k-th smallest element.

An example to show the correctness of the algorithm. Let us consider we have three arrays of size 7 with the following elements.

$$A : \text{int A[7]} = \{0, 4, 8, 12, 16, 20, 24\}$$
$$B : \text{int B[7]} = \{1, 5, 9, 13, 17, 21, 25\}$$
$$C : \text{int C[7]} = \{2, 6, 10, 14, 18, 22, 26\}$$

Let's say we want to find the 9th smallest element in the union of these three arrays. Hence we call the Find_the_kth_element with the following values of low and high.
low being the minimum of A[0],B[0],C[0] as 0
high being the maximum of A[6],B[6],C[6] as 26
Then the function call Find_the_kth_element is executed with low as 0 high as 26 and k as 9.

1st Call for Find_the_kth_element :-

- Mid is calculated 0+(26-0)/2=13
- The function Count is called three times for each of the arrays and count returns 4+4+3=11. Hence the function call: Find_the_kth_element(A,B,C,9,0,12) is executed

2nd Call for Find_the_kth_element :-

- Mid is calculated 0+(12-0)/2=6
- The function Count is called three times for each of the arrays and count returns 2+2+2=6. Hence the function call: Find_the_kth_element(A,B,C,9,7,12) is executed

3rd Call for Find_the_kth_element :-

- Mid is calculated 7+(12-7)/2=9
- The function Count is called three times for each of the arrays and count returns 3+3+2=8. Hence the function call: Find_the_kth_element(A,B,C,9,10,12) is executed

4th Call for Find_the_kth_element :-

- Mid is calculated 10+(12-10)/2=11
- The function Count is called three times for each of the arrays and count returns 3+3+3=9. Hence the function call: Find_the_kth_element(A,B,C,9,10,10) is executed

5th Call for Find_the_kth_element :-

- Mid is calculated 10+(10-10)/2=10
- The function Count is called three times for each of the arrays and count returns 3+3+3=9. Hence the function call: Find_the_kth_element(A,B,C,9,10,9) is executed

6th Call for Find_the_kth_element :-

- low becomes greater than the high hence the function returns with the answer as low=10