Theory Assignment-1: ADA Winter-2024

Aarav Mathur (2022005) Aniket Gupta (2022073)

28-01-2024

1 Preprocessing

None

2 Algorithm Description

This algorithm is based on a median-of-medians divide and conquer approach. In each iteration of the algorithm, we are essentially able to reduce half the search space of any one of the array. This is done till we reach the base case, where only one array remains, and at that point, we return the (modified) kth element in that remaining array.

3 Recurrence Relation

The recurrence relation for the algorithm is given by:

$$T(n_1, n_2, n_3) = T\left(\frac{n_i}{2}, n_j, n_k\right) + O(1)$$

where i, j, k can take any permutation of the values $1, 2, 3$.

4 Complexity Analysis

Note that computing the time complexity of the algorithm directly from the above recurrence may prove to be extremely difficult. Hence, we will compute the time complexity for the following simplified version of the recurrence:

$$T(n_1, n_2, n_3) = T\left(\frac{n_1}{2}, n_2, n_3\right) + O(1)$$

where $\mathrm{n}_1 = n_2 = n_3$

Hence, our recurrence becomes:

$$T(N) = T(\frac{5N}{6}) + O(1)$$

where $\mathrm{N} = \mathrm{n}_1 + n_2 + n_3$

Thus, from the above recurrence relation, we can say that the time complexity of our algorithm is $O(\log(N))$, since there is a very clear reduction of the search space by a constant factor of $\frac{1}{6}$ in each iteration.

5 Pseudocode

Algorithm 1 Median

function MEDIAN($arr, low, high$)
    $len \leftarrow high - low + 1$
    if $len = 0$ **then**
        return 0
    end if
    if $len \mod 2 = 0$ **then**
        return $arr[low + \frac{len}{2} - 1]$
    else
        return $arr[low + \frac{len}{2}]$
    end if
end function

Algorithm 2 Eliminate_right

function ELIMINATE_RIGHT($median, pivot, length, size, counter$)
    if $median > pivot$ **then**
        if $length \mod 2 = 0$ **then**
            $counter \leftarrow counter - \frac{length}{2}$
            $size \leftarrow size - \frac{length}{2}$
        else
            $counter \leftarrow counter - (\frac{length}{2} + 1)$
            $size \leftarrow size - (\frac{length}{2} + 1)$
        end if
    end if
    return $\langle size, counter \rangle$
end function

Algorithm 3 Eliminate_left

function ELIMINATE_LEFT($median, pivot, length, size, counter, k$)
    if $median < pivot$ **then**
        if $length \mod 2 = 0$ **then**
            $counter \leftarrow counter + \frac{length}{2}$
            $size \leftarrow size - \frac{length}{2}$
            $k \leftarrow k - \frac{length}{2}$
        else
            $counter \leftarrow counter + (\frac{length}{2} + 1)$
            $size \leftarrow size - (\frac{length}{2} + 1)$
            $k \leftarrow k - (\frac{length}{2} + 1)$
        end if
    end if
    return $\langle size, counter, k \rangle$
end function

Algorithm 4 find_k

```
function FIND_K(arr1, arr2, arr3, l1, r1, l2, r2, l3, r3, k, valid1, valid2, valid3)
    len1 ← 0, len2 ← 0, len3 ← 0, T ← 0, m1 ← 0, m2 ← 0, m3 ← 0
    M ← 0.0
    if valid1 then
        len1 ← r1 − l1 + 1
        m1 ← Median(arr1, l1, r1)
    end if
    if valid2 then
        len2 ← r2 − l2 + 1
        m2 ← Median(arr2, l2, r2)
    end if
    if valid3 then
        len3 ← r3 − l3 + 1
        m3 ← Median(arr3, l3, r3)
    end if
    T ← len1 + len2 + len3
    if m1 = 0 and m2 = 0 then
        M ← m3
    else if m2 = 0 and m3 = 0 then
        M ← m1
    else if m1 = 0 and m3 = 0 then
        M ← m2
    else if m1 × m2 × m3 = 0 then
        M ← (m1+m2+m3) / 2.0
    else
        arr ← [m1, m2, m3]
        Sort(arr)
        M ← arr[1]
    end if
    if k ≤ T/2 then
        if valid1 then
            ⟨T, r1⟩ ← Eliminate_right(m1, M, len1, T, r1)
        end if
        if valid2 then
            ⟨T, r2⟩ ← Eliminate_right(m2, M, len2, T, r2)
        end if
        if valid3 then
            ⟨T, r3⟩ ← Eliminate_right(m3, M, len3, T, r3)
        end if
    else
        if valid1 then
            ⟨T, l1, k⟩ ← Eliminate_left(m1, M, len1, T, l1, k)
        end if
        if valid2 then
            ⟨T, l2, k⟩ ← Eliminate_left(m2, M, len2, T, l2, k)
        end if
        if valid3 then
            ⟨T, l3, k⟩ ← Eliminate_left(m3, M, len3, T, l3, k)
        end if
        if l1 > r1 then
            valid1 ← false
        end if
        if l2 > r2 then
            valid2 ← false
        end if
        if l3 > r3 then
            valid3 ← false
        end if
        if valid1 and not valid2 and not valid3 then
            return arr1[l1 + k − 1]
        end if
        if valid2 and not valid1 and not valid3 then
            return arr2[l2 + k − 1]
        end if
        if valid3 and not valid1 and not valid2 then
            return arr3[l3 + k − 1]
        end if
        return find_k(arr1, arr2, arr3, l1, r1, l2, r2, l3, r3, k, valid1, valid2, valid3)
```

6 Assumptions

- The indexing is zero-based.
- No two arrays should have any common elements.
- Assume that we have only floor division in the code in case of integers, since the code is derived from cpp.

7 Proof of Correctness

In this algorithm we first compute the approximate median of all three arrays individually, followed by computing the median of those three medians. We then make a check, if the kth element is less than half the total number of elements in all three arrays. If it is, then we eliminate the right half of the array with the largest median, else we eliminate the left half of the array with the smallest median and accordingly adjust the value of k too. We then recursively call the function again with the updated values of the arrays and k. We keep doing this until we reach the base case, where we have only one array left. At this point we simply return the kth element of the array.

This algorithm might not work in the cases where any two arrays have common elements. This is because, the medians of two or more arrays might come out to be same, and the program might not be able to distinguish the array with the largest, or the array with the smallest median and enter into an infinite recursion call.

8 References and Collaborations

During the development and exploration of the algorithms presented in this document, the following references and collaborations were invaluable:

1. **CS Stack Exchange Discussion:**

   Title: *Kth smallest element in 2 sorted arrays*
   Link: https://stackoverflow.com/questions/4607945/how-to-find-the-kth-smallest-element-in-the-union-of-two-sorted-arrays

   The discussion on CS Stack Exchange provided insights and discussions related to the kth smallest element in 2 sorted arrays.

2. **YouTube Video Tutorial:**

   Title: *Algorithm Explanation: Find Kth Element in Two Sorted Arrays*
   Link: https://www.youtube.com/watch?v=nv7F4PiLUzohttps://www.youtube.com/watch?v=nv7F4PiLUzo

   This video tutorial offered a detailed explanation of the algorithm for finding the kth element in two sorted arrays.