

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) k th 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 $n_1 = n_2 = n_3$

Hence, our recurrence becomes:

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

where $N = n_1 + n_2 + n_3$

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

```
1: function MEDIAN(arr, low, high)
2:   len  $\leftarrow$  high - low + 1
3:   if len = 0 then
4:     return 0
5:   end if
6:   if len mod 2 = 0 then
7:     return arr[low +  $\frac{len}{2}$  - 1]
8:   else
9:     return arr[low +  $\frac{len}{2}$ ]
10:  end if
11: end function
```

Algorithm 2 Eliminate_right

```
1: function ELIMINATE_RIGHT(median, pivot, length, size, counter)
2:   if median > pivot then
3:     if length mod 2 = 0 then
4:       counter  $\leftarrow$  counter -  $\frac{length}{2}$ 
5:       size  $\leftarrow$  size -  $\frac{length}{2}$ 
6:     else
7:       counter  $\leftarrow$  counter - ( $\frac{length}{2}$  + 1)
8:       size  $\leftarrow$  size - ( $\frac{length}{2}$  + 1)
9:     end if
10:  end if
11:  return  $\langle$ size, counter $\rangle$ 
12: end function
```

Algorithm 3 Eliminate_left

```
1: function ELIMINATE_LEFT(median, pivot, length, size, counter, k)
2:   if median < pivot then
3:     if length mod 2 = 0 then
4:       counter  $\leftarrow$  counter +  $\frac{length}{2}$ 
5:       size  $\leftarrow$  size -  $\frac{length}{2}$ 
6:       k  $\leftarrow$  k -  $\frac{length}{2}$ 
7:     else
8:       counter  $\leftarrow$  counter + ( $\frac{length}{2}$  + 1)
9:       size  $\leftarrow$  size - ( $\frac{length}{2}$  + 1)
10:      k  $\leftarrow$  k - ( $\frac{length}{2}$  + 1)
11:    end if
12:  end if
13:  return  $\langle$ size, counter, k $\rangle$ 
14: end function
```

Algorithm 4 find k

```
1: function FIND_K(arr1, arr2, arr3, l1, r1, l2, r2, l3, r3, k, valid1, valid2, valid3)
2:   len1  $\leftarrow$  0, len2  $\leftarrow$  0, len3  $\leftarrow$  0, T  $\leftarrow$  0, m1  $\leftarrow$  0, m2  $\leftarrow$  0, m3  $\leftarrow$  0
3:   M  $\leftarrow$  0.0
4:   if valid1 then
5:     len1  $\leftarrow$  r1 - l1 + 1
6:     m1  $\leftarrow$  Median(arr1, l1, r1)
7:   end if
8:   if valid2 then
9:     len2  $\leftarrow$  r2 - l2 + 1
10:    m2  $\leftarrow$  Median(arr2, l2, r2)
11:  end if
12:  if valid3 then
13:    len3  $\leftarrow$  r3 - l3 + 1
14:    m3  $\leftarrow$  Median(arr3, l3, r3)
15:  end if
16:  T  $\leftarrow$  len1 + len2 + len3
17:  if m1 = 0 and m2 = 0 then
18:    M  $\leftarrow$  m3
19:  else if m2 = 0 and m3 = 0 then
20:    M  $\leftarrow$  m1
21:  else if m1 = 0 and m3 = 0 then
22:    M  $\leftarrow$  m2
23:  else if m1  $\times$  m2  $\times$  m3 = 0 then
24:    M  $\leftarrow$   $\frac{m1+m2+m3}{2.0}$ 
25:  else
26:    arr  $\leftarrow$  [m1, m2, m3]
27:    Sort(arr)
28:    M  $\leftarrow$  arr[1]
29:  end if
30:  if k  $\leq$   $\frac{T}{2}$  then
31:    if valid1 then
32:       $\langle T, r1 \rangle \leftarrow$  Eliminate_right(m1, M, len1, T, r1)
33:    end if
34:    if valid2 then
35:       $\langle T, r2 \rangle \leftarrow$  Eliminate_right(m2, M, len2, T, r2)
36:    end if
37:    if valid3 then
38:       $\langle T, r3 \rangle \leftarrow$  Eliminate_right(m3, M, len3, T, r3)
39:    end if
40:  else
41:    if valid1 then
42:       $\langle T, l1, k \rangle \leftarrow$  Eliminate_left(m1, M, len1, T, l1, k)
43:    end if
44:    if valid2 then
45:       $\langle T, l2, k \rangle \leftarrow$  Eliminate_left(m2, M, len2, T, l2, k)
46:    end if
47:    if valid3 then
48:       $\langle T, l3, k \rangle \leftarrow$  Eliminate_left(m3, M, len3, T, l3, k)
49:    end if
50:    if l1 > r1 then
51:      valid1  $\leftarrow$  false
52:    end if
53:    if l2 > r2 then
54:      valid2  $\leftarrow$  false
55:    end if
56:    if l3 > r3 then
57:      valid3  $\leftarrow$  false
58:    end if
59:    if valid1 and not valid2 and not valid3 then
60:      return arr1[l1 + k - 1]
61:    end if
62:    if valid2 and not valid1 and not valid3 then
63:      return arr2[l2 + k - 1]
64:    end if
65:    if valid3 and not valid1 and not valid2 then
66:      return arr3[l3 + k - 1]
67:    end if
68:    return find_k(arr1, arr2, arr3, l1, r1, l2, r2, l3, r3, k, valid1, valid2, valid3)
69:
```

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 k th 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 k th 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 k th 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=nv7F4PiLUzo><https://www.youtube.com/watch?v=nv7F4PiLUzo>

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