Theory Assignment-1: ADA Winter-2024

Rachit Arora (2022384) Mahi Mann (2022272)

1 Assumptions

1. k ranges from 1 to $3n$.
2. **Array indices are from** 0 **to** $n-1$**.**
3. **Passing in arrays or subarrays of an array into a function is a constant time operation, because in practice we can implement this by using start and end pointers instead. Arrays are passed directly in the pseudocode to avoid overly messy code.**
4. For an array A, $|A|$ is the size of array A. Similar to the previous point, accessing the size of an array is also a constant-time operation.

2 Preprocessing

None required.

3 Algorithm Description

We start with the arrays A, B and C as the main problem.

The algorithm is recursive. Consider the current subarrays to be S_0, S_1 and S_3, and the current element to be found as the k^{th} smallest.

Define $mid(S_i) = \lfloor \frac{n_i}{2} \rfloor$ where n_i is the size of the array S_i, where $i = 0, 1, 2$.

We have two cases, $\sum_{i=0}^{2} mid(S_i) < k$ and $\sum_{i=0}^{2} mid(S_i) \geq k$.

1. Let's consider the first case, $\sum_{i=0}^{2} mid(S_i) < k$.

   Without loss of generality, let $S_0[mid(S_0)]$ be the **minimum** of all $S_i[mid(S_i)]$ for $i = 0, 1, 2$.

   Then, all elements of $S_0[0, \cdots, mid(S_0)]$ can be written as the j^{th} smallest in $S_0 \cup S_1 \cup S_2$ where $j < k$ (refer to Proof of Correctness).

   So, our required element may not be in $S_0[0, \cdots, mid(S_0)]$ because they can be at most the $(k-1)^{\text{th}}$ smallest, but it is definitely elsewhere in the remaining subarray and 2 arrays.

   So, we recursively find the $(k - mid(S_0))^{\text{th}}$ smallest element in $S_0[mid(S_0) + 1, \cdots, n_0 - 1] \cup S_1 \cup S_2$.

2. Now for the latter case, $\sum_{i=0}^{2} mid(S_i) \geq k$.

   Without loss of generality, let $S_0[mid(S_0)]$ be the **maximum** of all $S_i[mid(S_i)]$ for $i = 0, 1, 2$.

   Then, all elements of $S_0[mid(S_0) + 1, \cdots, n_0 - 1]$ can be written as the j^{th} smallest in $S_0 \cup S_1 \cup S_2$ where $j > k$ (refer to Proof of Correctness).

   So, our required element may not be in $S_0[mid(S_0)+1, \cdots, n_0-1]$ because they can be at least the $(k+1)^{\text{th}}$ smallest, but it is definitely elsewhere in the remaining subarray and 2 arrays.

   So, we recursively find the k^{th} smallest element in $S_0[0, \cdots, mid(S_0)] \cup S_1 \cup S_2$.

The base case for the recursion is simple. If the arrays S_0, S_1 and S_2 all have a single element, then the elements smaller than them will be $k-1$ in total, so we return the minimum of these 3 elements.

Furthermore, for completeness, if at any point of the algorithm an array cannot be divided further, we only recurse further on arrays which can.

4 Recurrence Relation

Let $p = n_1 n_2 n_3$ be the current product of the sizes of A, B and C; n_1, n_2 and n_3 respectively. Since the size of one of the arrays in the next subproblem will be approximately halved, size of the next subproblem $p' = \frac{p}{2}$. The only serial (non-recursive) operation is comparing two elements in $A \cup B \cup C$, which can be done in $O(1)$ as defined, so the serial time taken is less than equal to c_0 for some positive real constant c_0.

Therefore, we have:

$$T(p) = T(p') + c_0$$
$$T(p) = T(\frac{p}{2}) + c_0$$

5 Complexity Analysis

The above recurrence relation can be expressed as

$$T(p) = aT(\frac{p}{b}) + k_0 p^c$$

where $a = 1, b = 2, c = 0$.

By Master's theorem, since $0 = \log_2 1$ and thus $c = \log_b a$,

$$T(p) = O(\log p)$$

Since $p = n_1 n_2 n_3$, $\log p = \log n_1 + \log n_2 + \log n_3 = 3 \log n$.

Therefore, the overall problem with 3 arrays of size n can be solved in $O(3 \log n) = O(\log n)$ time.

6 Pseudocode

Algorithm 1 KthSmallest

```
function KthSmallest(A, B, C, n, k)
    return KthIndex(A, B, C, k-1)
end function

function KthIndex(A, B, C, k)

    if A is empty and B is empty then return C[k]
    end if

    if A is empty and C is empty then return B[k]
    end if

    if C is empty and B is empty then return A[k]
    end if

    for I ∈ {A, B, C} do
        if I is empty then
            m_I ← ∞
            M_I ← −∞
        else
            m_I ← I[Mid(I)]
            M_I ← I[Mid(I)]
        end if
    end for

    M ← max(M_A, M_B, M_C)
    m ← min(m_A, m_B, m_C)

    n_tot ← Mid(A) + Mid(B) + Mid(C)

    if n_tot < k then

        if m_A = m then
            return KthIndex(A[Mid(A)+1, ···, |A| − 1], B, C, k - Mid(A) - 1)
        else if m_B = m then
            return KthIndex(A, B[Mid(B)+1, ··· ,|B| − 1], C, k - Mid(B) - 1)
        else
            return KthIndex(A, B, C[Mid(C)+1, ··· ,|C| − 1], k - Mid(C) - 1)
        end if

    else

        if M_A = M then
            return KthIndex(A[0, ··· , Mid(A)], B, C, k)
        else if M_B = M then
            return KthIndex(A, B[0, ··· , Mid(B)], C, k)
        else
            return KthIndex(A, B, C[0, ··· , Mid(C)], k)
        end if

    end if

end function
```

The Mid() function is defined as below:

```
function MID(Arr)
    s ← |Arr|
    return ⌊s/2⌋
end function
```

7 Proof of Correctness

The basic explanation and proof of how the algorithm works is provided in the algorithm description section itself.

Here are some of the more minute details:

- For the proof of $j < k$, consider the opposite. If $j \geq k$, the middle elements are at least the $(k+1)^{\text{th}}$ smallest, and therefore our assumption is false.
- For the proof of $j > k$ in case 2, consider the opposite. If $j \leq k$, the middle elements are at most the k^{th} smallest, and therefore our assumption is false.