Theory Assignment-1: ADA Winter-2024

Krishna Shukla (2022254) Kshitij (2022256)

28/01/2024

1 Credit

We advice some part of our algorithms to Mohammad massod and Aryan Singla(they asked us for the help

We also took some help of ChatGpt for just doing documentation and using latex part.Writting Code,thinking algo ,thinking recurrence relation,time and space complexity and thinking of proof is all of our hardwork(no help taken).

2 Preprocessing

Array contains the same number of element . Array indexing starts from 0 in pseudocode .

3 Algorithm Description

Base Cases:

- $k == 0$: Imagine you're searching for the smallest remaining element. Comparing the first elements of each non-empty array is like peeking at the beginning of each list. Whichever list starts with the smallest number is guaranteed to have the overall smallest element, even if you don't look further. That's why returning the minimum of those first elements works perfectly!
- Empty Arrays: If any list becomes empty during the search, it means all its elements have already been considered and discarded. We treat the remaining non-empty arrays as a simplified problem of finding the k-th element in two sorted lists. This makes sense because we just need to focus on the remaining elements, and a "helper function" specifically designed for two lists can handle this scenario efficiently.

Recursive Step:

- Dividing k: Instead of blindly splitting k in half, we divide it into three roughly equal parts using $\max(1, k//3)$. This ensures that each recursive call focuses on approximately one-third of the remaining elements, leading to a more balanced search and the desired logarithmic time complexity.
- Discarding Elements: We compare the middle values of each list, essentially peeking halfway through each remaining subset. Whichever list has the smallest middle value is guaranteed to have at least one-third of its elements smaller than that value. Therefore, we can safely discard those smaller elements because they cannot be the k-th smallest element overall. This significantly reduces the search space for the next recursive call!
- Recursive Calls: After discarding elements, we update k to reflect the remaining search space and recursively call the function. But, instead of searching the entire list again, we focus on the portion that wasn't discarded, which contains the k-th smallest element for sure. The choice of which k and which portion of the list to focus on depends on which middle value was smallest, effectively guiding the search towards the target element.

Key Concepts:

- *Divide and Conquer*: Break down the problem into smaller subproblems by strategically discarding elements and then recursively searching for the target within those subproblems.

- *Maintaining Order*: Discarding elements based on middle values ensures that the relative order of the remaining elements is preserved. This guarantees that the k-th smallest element will be found within the remaining portion of the combined list.
- *Logarithmic Time Complexity*: Discarding at least one-third of the elements in each recursive call leads to a logarithmic time complexity, making the algorithm efficient for large datasets.

4 Recurrence Relation

Time Complexity:

- *Divide and Conquer*: The algorithm divides the problem into smaller subproblems by discarding elements in each recursive call.
- *Discarding at least one-third*: In each recursive call, at least one-third of the remaining elements are discarded $\mathrm{T(n)=T(2n/3)+O(1)}$ as comparision can be done in $\mathrm{O(1)}$. This means the search space is reduced by a factor of 3 in each level of recursion.
- *Logarithmic reduction: This repeated division by* 2/3 *leads to a logarithmic pattern in the number of* recursive calls. In general, the time complexity of a divide-and-conquer algorithm where you divide the problem into roughly equal parts in each step is $O(\log(n))$, where n is the size of the initial problem.
- *Combined Size*: The total size of the search space is the combined length of the three arrays $(n1 + n2 + n3)$.
- *Result*: Therefore, the time complexity of the algorithm is $O(\log(n1 + n2 + n3))$, indicating that the number of steps grows logarithmically with the total size of the input arrays.

Space Complexity:

- *Recursive Call Stack*: The primary space usage comes from the recursive call stack. Each recursive call adds a new frame to the stack, storing function arguments and local variables.
- *Depth of Recursion*: The maximum depth of recursion is proportional to the logarithm of the total size of the input arrays, as each call reduces the search space by at least one-third.
- *Proportional Stack Size*: The call stack size is directly proportional to the depth of recursion.
- *Result*: Therefore, the space complexity is also $O(\log(n1 + n2 + n3))$, reflecting the logarithmic growth of the call stack with respect to the input size.

5 Complexity Analysis

Time Complexity:

- Same as the recurrence relation.

Space Complexity:

- Same as the recurrence relation.

6 Proof of Correctness

Theorem: Given three sorted arrays A, B, and C, and an integer k ($1 \leq k \leq n_{\text{total}}$, where n_{total} is the sum of the lengths of A, B, and C), the algorithm correctly finds the k-th smallest element among all three arrays.

Proof: The proof relies on induction over the value of k.

Base Case:

$k = 0$: In this case, the algorithm compares the first elements of each non-empty array and returns the minimum. This is clearly the smallest element because all other elements in the arrays are greater than or equal to it.

Empty Arrays: If any or all arrays become empty during the recursion, they are effectively skipped in the comparison. The remaining arrays are then handled by a helper function for two sorted arrays, which is itself proven to be correct. Therefore, the base cases correctly identify the smallest element.

Induction Hypothesis: Assume the algorithm correctly finds the $(k-1)$-th smallest element for all values of k between 1 and $n_{\text{total}} - 1$.

Induction Step: We need to prove that the algorithm correctly finds the k-th smallest element.

Divide and Conquer: The algorithm divides k into three roughly equal parts, $sub1$, $sub2$, and $sub3$. This ensures that each recursive call focuses on approximately one-third of the remaining elements.

Discarding Elements: Based on the middle values of each list, the algorithm discards up to $sub1$, $sub2$, or $sub3$ elements from each list, respectively. We need to prove that discarding these elements does not affect the k-th smallest element.

Preserving Relative Order: The key property here is that the algorithm only discards elements from each list that are guaranteed to be smaller than at least $(k - \text{sub1})$, $(k - \text{sub2})$, or $(k - \text{sub3})$ elements in that list. This is because the middle value is chosen such that at least one-third of the remaining elements are greater than or equal to it. Therefore, the discarded elements cannot be the k-th smallest element overall.

Recursive Calls: Each recursive call focuses on the remaining portion of the lists after discarding elements. By the induction hypothesis, we know that each individual call will correctly find the $(k-\text{sub1})$-th, $(k-\text{sub2})$-th, or $(k - \text{sub3})$-th smallest element within its respective portion.

Combining Results: Since the discarded elements were smaller than at least $(k - \text{sub1})$, $(k - \text{sub2})$, or $(k-\text{sub3})$ elements in their respective lists, the $(k-\text{sub1})$-th, $(k-\text{sub2})$-th, and $(k-\text{sub3})$-th smallest elements found by the recursive calls must be guaranteed to be greater than the discarded elements. Therefore, the k-th smallest element lies among the smallest element of the remaining elements in each list and the three elements found by the recursive calls. By comparing these four elements, the algorithm correctly identifies the true k-th smallest element.

Conclusion: By considering the base cases and establishing the induction step, we have shown that the algorithm correctly finds the k-th smallest element for all values of k between 1 and n_{total}. Therefore, the theorem is proven.

7 Pseudocode

Algorithm 1 Find k-th Element in Three Sorted Arrays

function FIND_KTH_ELEMENT(k, $arr1$, $start1$, $end1$, $arr2$, $start2$, $end2$, $arr3$, $start3$, $end3$)
  if $k == 0$ **then**
    return min(arr1[start1], arr2[start2], arr3[start3])
  end if
  $mid \leftarrow \max(1, k//3)$
  $sub1 \leftarrow \min(mid, \text{end1} - \text{start1})$
  $sub2 \leftarrow \min(mid, \text{end2} - \text{start2})$
  $sub3 \leftarrow \min(mid, \text{end3} - \text{start3})$
  if start1 == end1 **then**
    return FIND_KTH_ELEMENT_1(k, $arr2$, $start2$, $end2$, $arr3$, $start3$, $end3$)
  else if start2 == end2 **then**
    return FIND_KTH_ELEMENT_1(k, $arr1$, $start1$, $end1$, $arr3$, $start3$, $end3$)
  else if start3 == end3 **then**
    return FIND_KTH_ELEMENT(k, $arr1$, $start1$, $end1$, $arr2$, $start2$, $end2$)
  end if
  $index1 \leftarrow \text{start1} + \text{sub1} - 1$
  $index2 \leftarrow \text{start2} + \text{sub2} - 1$
  $index3 \leftarrow \text{start3} + \text{sub3} - 1$
  $value1 \leftarrow$ arr1[index1]
  $value2 \leftarrow$ arr2[index2]
  $value3 \leftarrow$ arr3[index3]
  if $value1 \leq value2$ and $value1 \leq value3$ **then**
    return FIND_KTH_ELEMENT($k - \text{sub1}$, $arr1$, $\text{start1} + \text{sub1}$, end1, $arr2$, start2, end2, $arr3$, start3, end3)
  else if $value2 \leq value1$ and $value2 \leq value3$ **then**
    return FIND_KTH_ELEMENT($k - \text{sub2}$, $arr1$, start1, end1, $arr2$, $\text{start2} + \text{sub2}$, end2, $arr3$, start3, end3)
  else
    return FIND_KTH_ELEMENT($k - \text{sub3}$, $arr1$, start1, end1, $arr2$, start2, end2, $arr3$, $\text{start3} + \text{sub3}$, end3)
  end if
end function
function FIND_KTH_ELEMENT_1(k, $array1$, $start1$, $end1$, $array2$, $start2$, $end2$)
  if $k == 0$ **then**
    return min(array1[start1], array2[start2])
  end if
  $mid \leftarrow \max(1, k//2)$
  $sub1 \leftarrow \min(mid, \text{end1} - \text{start1})$
  $sub2 \leftarrow \min(mid, \text{end2} - \text{start2})$
  $index1 \leftarrow \text{start1} + \text{sub1} - 1$
  $index2 \leftarrow \text{start2} + \text{sub2} - 1$
  $value1 \leftarrow$ array1[index1]
  $value2 \leftarrow$ array2[index2]
  if $value1 \leq value2$ **then**
    return FIND_KTH_ELEMENT_1($k - \text{sub1}$, $array1$, $\text{start1} + \text{sub1}$, end1, $array2$, start2, end2)
  else
    return FIND_KTH_ELEMENT_1($k - \text{sub2}$, $array1$, start1, end1, $array2$, $\text{start2} + \text{sub2}$, end2)
  end if
end function