Theory Assignment-1: Rubric

Kuber Budhija

1 Preprocessing and Notation

Since the arrays are already in sorted order, no preprocessing is required. A: Array 1, B: Array 2, C: Array 3 , i_1 : elements behind insertion point in array 1, i_2 : elements behind insertion point in array 2, i_3 : elements behind insertion point in array 3, x : middle point whose insertion point is required to be found

This part contains no score, students are allowed to define their notations anyhwere in their solution

2 Algorithm Description

Since initially all three arrays are of same size, we'll choose A and find the insertion point (The position at which the element exists or should exist in a sorted array – found using binary search) of the middle element of A in both A and B, The current situation is as follows:

mentioning use of finding insertion points will fetch 2 points

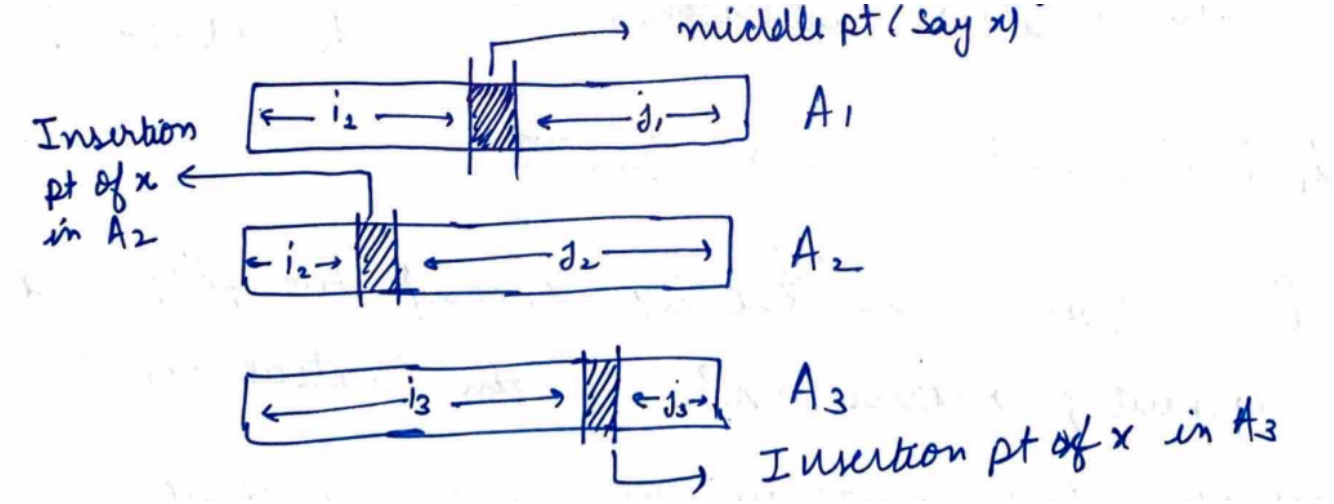


Figure 1: Enter Caption

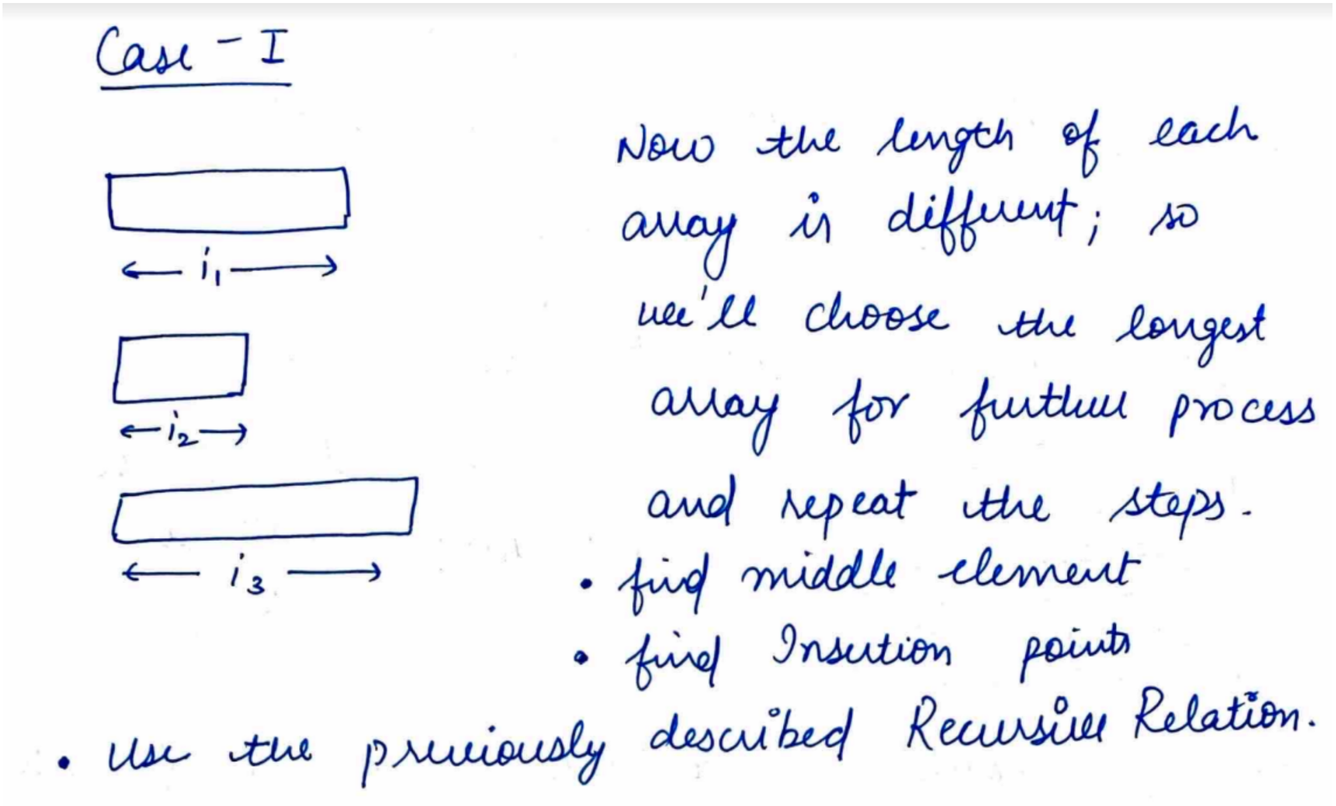
Since A, B and C are sorted; we know that there exists $i_1 + i_2 + i_3$ elements less than 'x' and $j_1 + j_2 + j_3$ elements greater than 'x' ,

since we are required to find kth smallest element; we can divide this problem into 3 furthur subcases,

CASE1 : When $(i_1 + i_2 + i_3) > k$

since $(i_1 + i_2 + i_3)$ is greater than and there exists these many numbers less than 'x', the kth smallest number also lies within these $(i_1 + i_2 + i_3)$ numbers (Think!)

obviously $(i_1 + i_2 + i_3) > k$ means ; $(i_1 + i_2 + i_3)$ = k + c(some constant), so our main problem now changes into this subproblem :



Case - I

Now the length of each array is different; so we'll choose the longest array for further process and repeat the steps.

- find middle element
- find Insertion points
- Use the previously described Recursive Relation.

Figure 2: Enter Caption

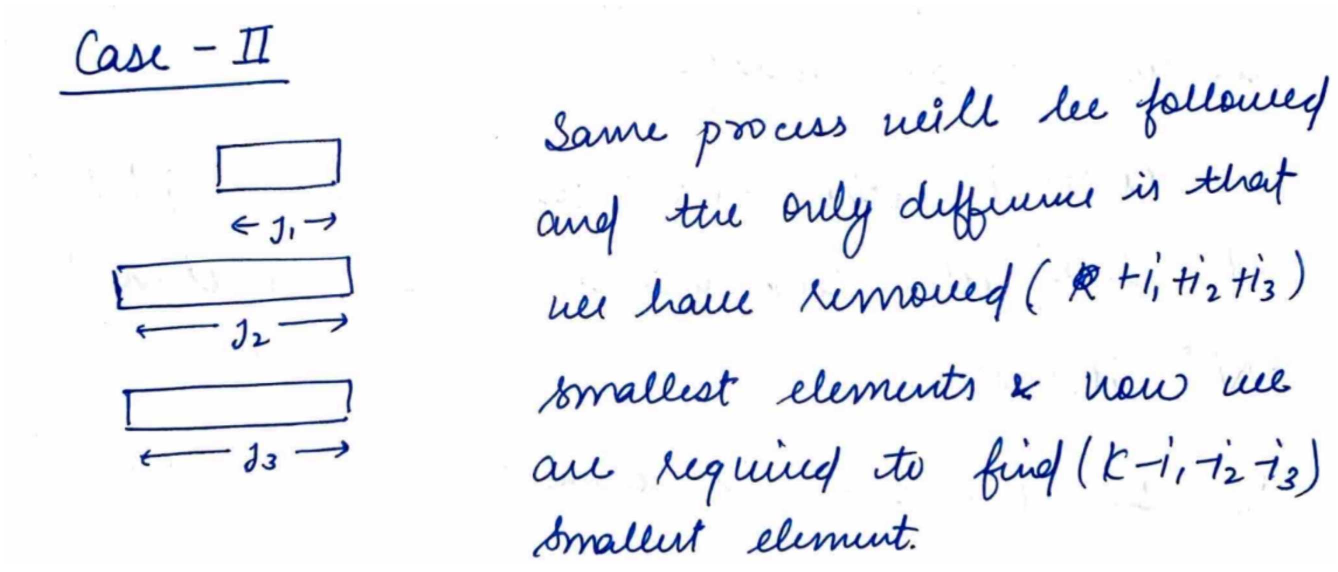
we'll now choose the left ends for further solution as k-th smallest number lies there (simple inference from the fact that arrays are sorted)

CASE2 : When $(i_1 + i_2 + i_3) < k$

now we know that we have found $(i_1 + i_2 + i_3)$ elements which are less than 'x', but that is less than k, therefore there exits k - $(i_1 + i_2 + i_3)$ elements between k-th smallest number and 'x', so we for sure now know that correct solution exists in the right parts of our arrays, and the scenario is as follows:

Choose the right part of array as we have covered $(i_1+i_2+i_3)$ small elements behind 'x' but we are still k-$(i_1+i_2+i_3)$ elements behind from what we want so we'll update k = k-$(i_1 + i_2 + i_3)$ and follow the same process (for better understanding take a look at pseudo code provided)

missing updation of k will fetch a score of negative 1


Case - II
$\leftarrow j_1 \rightarrow$
$\longleftarrow j_2 \longrightarrow$
$\longleftarrow j_3 \longrightarrow$

Same process will be followed and the only difference is that we have removed $(R + i_1 + i_2 + i_3)$ smallest elements & now we are required to find $(k - i_1 - i_2 - i_3)$ smallest element.

Figure 3: Enter Caption

CASE3 : When $(i_1 + i_2 + i_3) == k$ Voila! we have luckily reached the solution point (or maybe after many iterations but we have), since $(i_1 + i_2 + i_3)$ is K this means we have attained the point where all the elements smaller than it sums up to the count of k and hence this x is the next smallest element after k small elements.

breaking the problem into these three cases will fetch 3 points

3 Recurrence Relation

if $(i_1 + i_2 + i_3) > k$:
choose left parts of all these arrays
if $(i_1 + i_2 + i_3) < k$:
choose right parts of all these arrays and update k = k-$(i_1 + i_2 + i_3)$
if $(i_1 + i_2 + i_3) = k$:
return 'x'

4 Complexity Analysis

Let's take the number of arrays to be N. We'll be doing generic time complexity analysis and will take $N = 3$ for this case.

We'll assume the worst of all cases, by choosing a particular insertion point. We could only remove half of the elements of the chosen array and none from others. Even in that case, we'll be able to remove at least half of the total elements in N steps. (Think!)

Identifying worst case scenario will fetch 0.5 points, and mentioning the halving in N steps will fetch 0.5 points

So, in the worst possible scenario, we'll be able to remove half of the elements in N iterations, i.e., $X \to \frac{X}{2} \to \frac{X}{4} \to \ldots \to 1$ (till only one element is left). The height of the recursion tree can be found as follows:

$$\frac{X}{\mathrm{pow}(2,h)} = 1 \quad \Rightarrow \quad \mathrm{pow}(2,h) = X \quad \Rightarrow \quad h = \frac{\log X}{\log 2}$$

height calculation - 1 point

(Note: I am using the `pow(2,s)` convention of Python because it was taught in the IP course, and h represents the height of the recursion tree.)

Now, the total time taken is given by:

$$\text{Total time taken} = (\text{time taken at each step}) \times (\text{height of the recursion tree})$$

Now let's focus on time taken at each step:

we are trying to find insertion point for all the arrays finding the middle elements is O(1) step and finding insertion point is O(logX) as it uses same mechanism as binary search (you need not explain the time complexity of binary search and just mentioning it works!)

for N = 3, upper bound of time taken at every step is O(logX) (Mention the reasoning behind upper bound being O(logX), where X = number of elements, and after every iteration number of elements is reducing, and when it comes to time taken it is applied on each array separately and so each array will have X/3 elements initially given all arrays have same elements, so time taken is definitely less than O(logX))

explanation of time taken at each step - 1 point

so total time taken = O(log(X)*log(X))

not mentioning total complexity will attract a score of negative 0.5

5 Pseudocode

Algorithm 1 Insertion Point Algorithm

```
function INSERTION_PT(array, point)
    $high \leftarrow$ array.length $- 1$
    $low \leftarrow 0$
    while ($low \leq high$) do
        $mid \leftarrow low + \frac{(high-low)}{2}$
        if ($array[mid] =$ point) then
            return $mid$
        else if ($array[mid] >$ point) then
            $high \leftarrow mid - 1$
        else
            $low \leftarrow mid + 1$
        end if
    end while
    return $low$
end function
```

Correct pseudocode of insertion point fetches 1.5 points, else 0 [no partial marking].

Algorithm 2 Your Algorithm

```
function SOLVE(A, B, C, k)
    if (A.len $\geq$ B.len and A.len $\geq$ C.len) then
        $x \leftarrow$ A.len()/2
        $i1 \leftarrow$ Insertion_pt(B, $A[x]$)
        $i2 \leftarrow$ Insertion_pt(C, $A[x]$)
        if ($i1 + i2 + x = k$) then
            return $A[x]$
        else if ($i1 + i2 + x < k$) then
            SOLVE(A[A.len()/2:], B[i1:], C[i2:], $k - i1 - i2 -$ A.len()/2)
        else
            SOLVE(A[:A.len()/2], B[:i1], C[:i2], $k$)
        end if
    else if (B.len $\geq$ A.len and B.len $\geq$ C.len) then
        $x \leftarrow$ B.len()/2
        $i1 \leftarrow$ Insertion_pt(A, $B[x]$)
        $i2 \leftarrow$ Insertion_pt(C, $B[x]$)
        if ($i1 + i2 + x = k$) then
            return $B[x]$
        else if ($i1 + i2 < k$) then
            SOLVE(B[B.len()/2:], A[i1:], C[i2:], $k - i1 - i2 -$ B.len()/2)
        else
            SOLVE(B[:B.len()/2], A[:i1], C[:i2], $k$)
        end if
    else if (C.len $\geq$ B.len and C.len $\geq$ A.len) then
        $x \leftarrow$ C.len()/2
        $i1 \leftarrow$ Insertion_pt(B, $C[x]$)
        $i2 \leftarrow$ Insertion_pt(A, $C[x]$)
        if ($i1 + i2 + x = k$) then
            return $C[x]$
        else if ($i1 + i2 + x < k$) then
            SOLVE(C[C.len()/2:], B[i1:], A[i2:], $k - i1 - i2 -$ C.len()/2)
        else
            SOLVE(C[:C.len()/2], B[:i1], A[:i2], $k$)
        end if
    end if
end function
```

Correct explanation of pseudocode of solve will fetch 4 points, after that every 2 mistakes will attract -1.

6 Rubric guidelines for solutions

- If a solution's runtime is better than O(n) but worse than O(logn*logn), the solution will only fetch 12 marks with breakdown as follows:
  - +4 for correct pseudocode else +0
  - +4 for correct algorithm explanation, partial marks here are subjective
  - +4 for time complexity analysis breakdown as : +2 for step time and +2 for finding height and calculating total time.
- any linear time solution will fetch +2
- worse than linear time award 0.

7 An $O(\log n)$ solution to the problem

Solution: We provide a $O(\log n)$ running time solution for the given problem. In this solution, we do not use the binary search feature. Rather, we efficiently remove at least half of the size of one among the three arrays at each step.

7.1 Assumptions and notations

We keep the array indexing as per the question; that is, the starting index is one, and the ending index is n for each array. We consider that $n \geq 1$. Also, consider k to be a valid input, that is, $1 \leq k \leq |A \cup B \cup C|$.

Given any array, say $X[i \cdots j]$ with starting index i and ending index j; $mid(X)$ defines the middle element of X. To be precise,

$$mid(X) = X[m], \text{ where } m = \left\lfloor \frac{j-i+1}{2} \right\rfloor$$

Moreover, $\mathcal{I}_{mid}(X) = m$ denotes the middle index of the array

7.2 Algorithm Description

We write the broad idea of the algorithm in three simple steps.

1. Let $X_{\min} \in \{A, B, C\}$ be the array with the minimum of the middle elements of three arrays. Similarly, let $X_{\max} \in \{A, B, C\}$ be the array with the maximum of the middle elements of three arrays. Note, $X_{\min}$ and $X_{\max}$ both are non empty. Thus at least one among A, B, C must be non-empty.
2. If k is greater than half of the total elements in $A \cup B \cup C$, then we remove the elements in the first half of $X_{\min}$. We also update the value of k as we have removed $\lfloor \frac{n}{2} \rfloor$ elements from $A \cup B \cup C$ which were smaller than k (proof of the elements being smaller (or equal) than k is given in lemma 1).

   Otherwise, if k is less than half (or equal to) of the total elements in $A \cup B \cup C$, then we remove the elements in the last half of $X_{\max}$. Here, we don't need to update the value of k because we have removed $\lfloor \frac{n}{2} \rfloor$ elements from $A \cup B \cup C$ which were greater than k (see lemma 2).
3. Repeat steps 1 and 2 with the updated array until $A \cup B \cup C$ contains only one element. Report that element as k-th element.

7.3 Algorithm

Now we give further technical details. All the arrays are of length n. Following is the algorithm.

```
1. (a) $X_{\min} = \arg\min_{X \in \{A,B,C\}} \{mid(X)\}$
   (b) $X_{\max} = \arg\max_{X \in \{A,B,C\}} \{mid(X)\}$
2. (a) If $k > (\mathcal{I}_{mid}(A) + \mathcal{I}_{mid}(B) + \mathcal{I}_{mid}(C))$ then
         i. Set $X_{\min} \leftarrow X_{\min}[\mathcal{I}_{mid}(X_{\min}) + 1 \cdots n]$ when $\mathcal{I}_{mid}(X_{\min}) > 0$;
            Set $X_{\min} \leftarrow \emptyset$ otherwise
        ii. Set $k \leftarrow k - \mathcal{I}_{mid}(X_{\min})$
   (b) Else
         i. Set $X_{\max} \leftarrow X_{\max}[1 \cdots \mathcal{I}_{mid}(X_{\max})]$ when $\mathcal{I}_{mid}(X_{\max}) > 0$;
            Set $X_{\max} \leftarrow \emptyset$ otherwise
3. Repeat step 1 and step 2 until $(\mathcal{I}_{mid}(A) + \mathcal{I}_{mid}(B) + \mathcal{I}_{mid}(C) = 1)$
4. Output $X[1]$ where $X = A \cup B \cup C$
```

7.4 Proof of correctness

Lemma 1 *When $k > (\mathcal{I}_{mid}(A)+\mathcal{I}_{mid}(B)+\mathcal{I}_{mid}(C))$, then the k-th element can not belong to $X_{min}[1\cdots\mathcal{I}_{mid}(X_{\min})]$*

Proof: For contradiction, let k-th element lie in the subarray. Now observe that,

$$\mathcal{I}_{mid}(X_{\min}) \leq \left\lfloor \frac{|A \cup B \cup C|}{2} \right\rfloor$$

Again, we have,

$$\left\lfloor \frac{|A \cup B \cup C|}{2} \right\rfloor \leq (\mathcal{I}_{mid}(A) + \mathcal{I}_{mid}(B) + \mathcal{I}_{mid}(C)) + 1 \leq k$$

This implies that since any element in $X_{min}[1\cdots\mathcal{I}_{mid}(X_{\min})]$ must lie in the first half of $A \cup B \cup C$, hence they are smaller than k-th smallest element. □

Lemma 2 *When $k < (\mathcal{I}_{mid}(A)+\mathcal{I}_{mid}(B)+\mathcal{I}_{mid}(C))$, then the k-th element can not belong to $X_{\max}[\mathcal{I}_{mid}(X_{\max})+1\cdots n]$*

Proof: The proof is similar to the proof of lemma 1 □

7.5 Running time

Lemma 3 *The algorithm converges after $O(logn)$ number of steps.*

Proof: Lemma 2 and lemma 1 ensure that in every iteration, any one of the arrays A, B, C is always getting halved from its previous size. Now, one array can get halved by at most $O(\log n)$ times before it becomes empty because the initial size is n. Once an array decreases, it never increases in any step of the algorithm. Each array shrinks at most $O(\log n)$ time, and before every shrinkage, we access the middle elements of three arrays and find the maximum and minimum among them, which takes $O(1)$. Thus, the running time is $O(\log n)$. □