Q1 [redacted] MOKSH SHUKLA 180433

Algorithm

→ we need to find 2 local maximas corresponding to i,j.

→ we can use function local maxima similar to local-minima discussed in class to find local maxima.

→ we find 1st local maxima using the above described function.
Suppose it occurs at k.
Then again g will call the same local maxima function to find local maxima in the array from (k+1) to n.

→ Thus, 2 calls will give us the 2 index of local maxima.

Pseudo Code :

```
local-maxima-array(A) {
        L←0; R←n-1;
        present ← false;
        while (not present) {
            mid ←(L+R)/2;
            if (mid is local maxima) {
                present ← True;}
            else if (A[mid+1] >A[mid]) { L←mid+1';}
            else if (A[mid+1]<A[mid]) { R← mid -1;}
        }
        return mid;
        }

Find-index(A){
        itr1 ← local-maxima-array(A);
        a ← local-maxima-array(A[0---itr1]);
        b ← local-maxima-array(A[itr1---n-1]);
        if (a == itr1) {itr2 ← b;}
        else { itr2 ← a;}
        return (itr1, itr2)
        }
```

returns 2 outputs

Time Complexity :

→ As discussed in lectures local maxima will take $O(\log n)$ time.

→ other computations occur in $O(1)$

overall = $O(\log n)$

Space Complexity : $O(1)$ if ~~array~~ no original array not considered

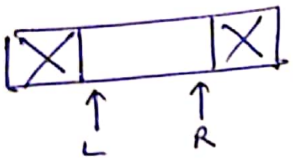
$O(n)$ if " " considered

Proof of Correctness : (similar to local minima one in class)

Assertion P[i] : At end of i^{th} iteration, local maxima exists in A[L,R]

Base Case : A local maxima always exists in A[0,n-1]

Assume P[i-1] is true.



→ mid = $\left(\frac{L+R}{2}\right)$. If A[mid] is maxima return it.

→ if A[mid+1] > A[mid] ⇒ L ← mid+1
so new subarray is [mid+1, R] which has a maxima

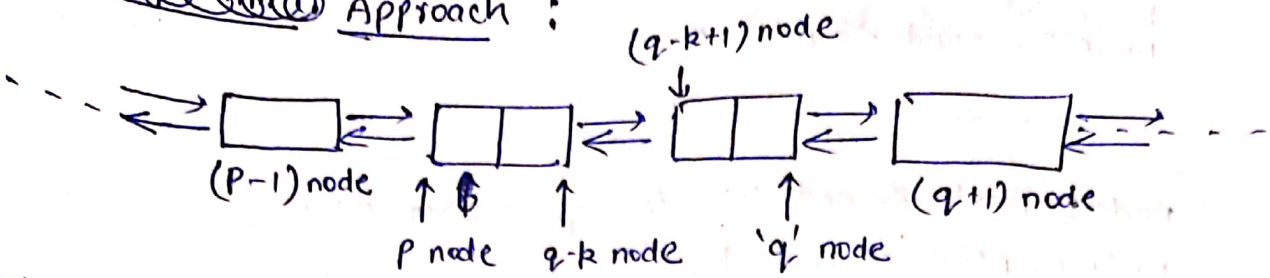
→ if A[mid+1] < A[mid] ⇒ R ← mid-1
so new subarray is [mid-1, R] which has a maxima.

Hence, P[i] is true as we are sure to have a maxima in [L,R] at i^{th} iteration depending on the case we encounter.

Correctness proved.

Q2

Algorithm Approach :



- We need to rotate elements b/w p & q by 'k' places.
- As we observe, nodes before p and after q are not touched so the connections b/w these nodes remain same.
- Relative order b/w $(a_p, a_{p+1} \ldots a_{q-k})$ is same as well as b/w $(a_{q-k+1}, \ldots a_q)$
- So we just need to change pointers b/w some nodes. So break connection b/w (q-k) & (q-k+1) and connect (q-k+1) to (p-1). Similarly connect (q+1) to (q-k). Also connect nodes p & q together.
- Thus, our list is rotated.

Pseudo Code :

```
lookup-node(head, loc) {
        temp ← head;
        for(i=1 to loc) { temp ← temp.next; }
        return temp;
    }

rotate_list(head, p, q) {
    node 1 ← node_lookup(head, p-1);
    node 2 ← node 1.next;
    node 3 ← node_lookup(head, q-k);
    node 4 ← node 3.next;
    node 5 ← node_lookup(head, q);
    node 6 ← node 5.next;
```

node 1. next ← node 4; } (p-1) ⇔ (q-k+1)
node 4. prev ← node 1;

node 3. next ← node 6; } (q+1) ⇔ (q-k)
node 6. prev ← node 3;

node 5. next ← node 2; } p ⇔ q
node 2. prev ← node 5;

3

Time Complexity :

node-lookup has a single loop ⇒ O(n) — 3 times call made in rotak-list but separately
All other operations in rotate-list ⇒ O(1)

overall = O(n)

Space Complexity : O(1)

Q3 : Algorithm :

① Find the median of sorted arrays A and B . Let their medians be medA and medB .

② If medA > medB , Then median is present b/w first element of A and medA or b/w (medB+1) to end element of B in respective subarrays .

③ If medA < medB , then case is opposite to defined above

④ If medA == medB we return directly

This way we can keep on reducing the array sizes and finally reach a solution . ⑤ This way we keep on reducing array sizes . till we reach array with n=1 or 2

if n==1 ⇒ median ← (A[0]+B[0])/2

if n==2 ⇒ median ← $\frac{\max(A[0],B[0]) + \min(A[1],B[1])}{2}$

⑤ using n=1 or n=2 we finally find our median (A+B) combined .

Pseudo Code :

```
median_array (A, B, n) {
    if (n==1) { return (A[0]+B[0])/2 ; }
    if (n==2) { return ( max(A[0],B[0]) + min(A[1],B[1]) ) / 2 ; }

    medA ← median in array A ;
    medB ← median in array B ;
    if (medA > medB) {
        if (n is even) { return median_array ( B+n/2 -1 , A , n/2 +1 ); }
        else { return median_array (B+n/2 , A , n/2) ; }
    }
    else { if (n is even) { return median_array (A+n/2 -1 , B , n/2 +1 ); }
           else { return median_array (A+n/2 , B , n/2) } }
```

Time Complexity : As the array gets divided into smaller versions at each step, it follows divide & conquer paradigm and hence time complexity is $O(\log n)$

Space Complexity ⇒ $O(1)$ → no new array formed

Proof of Correctness :

Assertion P[i] : Subarray at i^{th} iteration always has the median

Base Case : When we have $n = 2$

we say median $= \frac{\max(A[0], B[0]) + \min(A[1], B[1])}{2}$

because $\min(A[0], B[0])$ element will be at 0^{th} index of combined array and $\max(A[1], B[1])$ element will be at $(n-1)^{th}$ index of combined array.

Assume P[i-1] is true.

At this step we have 2 subarrays of equal size. Let us call them X, Y. Let their medians be medX, medY. Size of subarry = n

Case I If medX > medY

since medY is smaller than medX, medY will come before medX in combined array.

Max. elements that are smaller than medY is $(\frac{n}{2} - 1)$ elements each in array X and Y.

$(n-2)$ is max no. of total elements in combined array.

We can clearly observe that since these elements are smaller than medY they do contribute to median of combine array.

By surity we can remove $(\frac{n}{2} - 1)$ elements from Y that are smaller than medY. This surity does not exist exist for X.

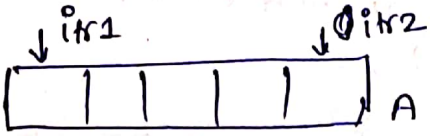
As we removed $(\frac{n}{2} - 1)$ elements from start of Y, we can also removed $(\frac{n}{2} - 1)$ elements from last of X by similar argument.

∴ our median still lies in given subarrays

Case II) Similar to Case I. only change is arguments for X become that of Y and vice-versa (medX < medY)

P[i] Proved

Q4 **Algorithm :**



$a^2 = bx + cy$

① Take 2 iterators itr1 and itr2 where itr1 iterates from 0 to n-1 and itr2 in reverse from (n-1) to 0.
itr1 corresponds to providing 'x' and itr2 corresponds to providing 'y' from array A. We do this since our array is sorted.

② Compute $bx + cy$ and compare with a^2. Now if $a^2 = bx + cy$, return TRUE. If at one iteration $a^2 > bx + cy$ move itr1 → itr1+1. If at any iteration $a^2 < bx + cy$ move itr2 → itr2-1

③ Repeat process in ② till itr1 < itr2. (itr1 ≠ itr2 since we want distinct)

④ If any possible pair of x, y satisfies $a^2 = bx + cy$, return TRUE. Else FALSE

⇒ **Pseudo Code :**

```
Pair_num(A, a, b, c) {
        itr1 ← 0 ; itr2 ← n-1 ;
        while( itr1 < n and itr2 >= 0) {
            x ← A[itr1] ;
            y ← A[itr2] ;
            if (a^2 == bx + cy) { return TRUE ; }
            else if ( a^2 < bx + cy) { itr2 ← itr2 - 1 ; }
            else if ( a^2 > bx + cy) { itr1 ← itr1 + 1 ; }
        }
        return FALSE ;
}
```

Time Complexity : we iterate only once through array A, since we have iterators running simultaneously.
Hence, O(n)

Space Complexity : O(1) if array A not considered
O(n) if array A is considered.

⇒ Proof of Correctness :

Assertion P[i] : At end of iteration i, we have $bx+cy \neq a^2$ for $x \in [0, itr1-1]$ & $y \in [itr2+1, n]$ OR $x \in [itr2+1, n]$ & $y \in [0, itr1-1]$. i.e we do not have a solution in given range for x,y

Base Case : itr1 = 0 & itr2 = n-1, entire array available for x, y. Hence holds true.

Assume P[i-1] to be true.

→ Before i^{th} iteration starts, we have $x = A[itr1]$ and $y = A[itr2]$

→ if $a^2 = bx+cy$ we return true

→ if $a^2 > bx+cy$, we do itr1 → itr+1 to increase value of RHS (since array is sorted). In doing so we increase our chances of getting an equality without missing out on any solution.

→ if $a^2 < bx+cy$, we do itr2 → itr2-1, to decrease value of LHS (sorted array). In doing so decrease value of y, in an attempt to achieve equality. In doing so we never miss out on any possible value of y which can make equality true.

Hence, we move to i^{th} iteration.

Correctness proved

MOKSH SHUKLA
180433

TA-1
PART 2

Q5 (a) $\min(n^2, 10^{12}) = O(1)$ prove

→ $\min(n^2, 10^{12})$ can be calculated by 2 step conditional operation :
(Assume n to be large as we are looking at asymptotic time complexity)

if ($n^2 > 10^{12}$) {ans = n^2 }
else { ans = 10^{12} }

constant time in O(1) time

⇒ Hence, we can say $\min(n^2, 10^{12})$ can be calculated in O(1) time.

(b) $n^2 + n\log n = O(n^2)$ prove

we know $f(n) = O(g(n))$ if $f(n) \le c\,g(n)$ for $n > n_0$ - (*)

$n\log n \le n^2$ { $\lim_{n\to\infty} \frac{n\log n}{n^2} \to 0$ } $\forall\ n \ge 0$

adding n^2 on both sides

$n^2 + n\log n \le 2n^2 \quad \forall\ n \ge 0$

So, we can say $n^2 + n\log n = O(n^2)$ by (*) where
$c = 2,\ n_0 = 0$

(c) $n^3 + 3n^2 + 8 \neq O(n^2)$

Let $f(n) = n^3 + 3n^2 + 8$; $g(n) = n^2$

we know : $n^3 > n^2 \quad \forall\ n \ge 1$

$n^3 + 3n^2 \ge 4n^2 \quad \forall\ n \ge 1$

adding a constant 8 to LHS does not change inequality sign

$n^3 + 3n^2 + 8 \ge 4n^2 \quad \forall\ n \ge 1$ ⇒ $n^3 + 3n^2 + 8 \ge n^2 \quad \forall\ n \ge 1$

∴ $n^3 + 3n^2 + 8$ is not bounded by n^2

if $f(n) = O(g(n))$ we need $f(n) \le cg(n)$ for $n > n_0$

but we have the relation in this question as :

$f(n) \ge cg(n)$ for $n > n_0$ where our $c = 4$ and $n_0 = 1$

Hence, proved $n^3 + 3n^2 + 8 \neq O(n^2)$

Hence Proved $n^3 + 3n^2 + 8 \neq O(n^2)$

(d) $4^n \neq O(2^n)$ prove

Let $f(n) = 4^n$; $g(n) = 2^n$

$4^n \geq 2^n \quad \forall \quad n \geq 1 \Rightarrow 4^n$ not bounded by 2^n

~~case in previous question, here also $f(n) \geq c\,g(n)$ with c=1 and n=0~~

Hence, $\boxed{4^n \neq O(2^n)}$.

(e) $\log(n!) = O(n \log n)$ prove

$\log n! = \log(1) + \log(2) + \cdots + \log(n)$

we can write :

$\log(1) \leq \log(n)$

$\log(2) \leq \log(n)$

$\vdots$

$\log(n-1) \leq \log(n)$

$\log(n) = \log(n)$

$\forall\ n \geq 1$

Adding these up

$\log(1) + \log(2) + \cdots \log(n) \leq n \log(n) \quad \forall\ n \geq 1$

$\log(n!) \leq n \log n \quad \forall \quad n \geq 1$

Similar to $f(n) \leq c\,g(n)$ for $n > n_0$

where $f(n) = \log(n!)$

$g(n) = n \log n$

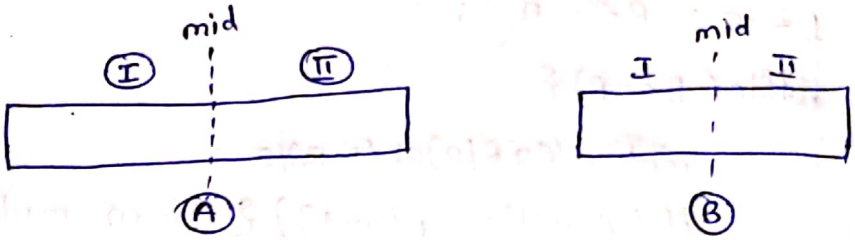
$c = 1$; $n_0 = 1$

Hence $\boxed{\log n! = O(n \log n)}$.

MOKSH SHUKLA 180433

Q6) Since, we need to design O(logn) algorithm we will need to somehow use divide and conquer approach.

Algorithm :



① Compare A[mid] and B[mid]. If A[mid] == B[mid], then our search for index 'k' is done. Hence, return (k = mid).

② If ① does not hold true then consider 2 cases:

Case-I) If A[mid] is greater than B[mid], then A[k] = B[n-1-k] occurs when index 'k' is in (AI) and 'n-1-k' index is in (BII)

Case-II) If A[mid] is less than B[mid], then A[k] = B[n-1-k] occurs when 'k' is in (AII) and 'n-1-k' is in (BI).

③ One of the 2 cases from above holds and we get 2 subarrays of A and B. We now again compare the middle of both subarrays and repeat the check of Case I and Case II and then further get 2 arrays of reduced size..

④ Repeat the process described above to keep dividing the arrays.
At the end of this divison, we are left with just 2 elements, one from A & B each.

⑤ Compare the remaining elements and if they are equal return the index 'k', else return -1.

⇒ Pseudo Code:

```
Index_Equal(A,B) {
    L ← 0 ; R ← n-1 ;
    while( L<=R) {
        mid = (L+R)/2 ;
        if ( A[mid] == B[mid] ) { return mid ; }
        else if ( A[mid] < B[n-1-mid]) { R ← mid-1 ; }
        else if ( A[mid] > B[n-1-mid]) { L ← mid+1 ; }
    }
    return -1 ;
}
```

⇒ Time Complexity : Size of array getting half at each iteration.

So, recurrence relation : $T(n) = T(n/2) + c$.

which will finally give $O(\log n)$ as time complexity

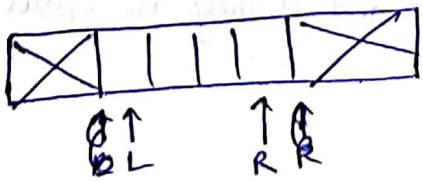
⇒ Space Complexity : Since, we not created any extra element or array or storage element, we haven't used any extra space.

So, space complexity is $O(1)$.

If we consider already provided arrays A & B, then space complexity will be $O(n)$.

⇒ PROOF OF CORRECTNESS :

Assertion @ $P(i)$: $K \in [L,R]$ at every iteration step, in both A & B



Base Case : when $i=0$; L is at start i.e $L=0$ and R is at end i.e. $R = n-1$

→ Assuming, P[i-1] to be true

→ Since P[i-1] is true, at the end of this iteration Then range for $K \in [L, R]$

→ $mid = \frac{L+R}{2}$. If A[mid] == B[mid], we return K

→ if A[mid] > B[mid], Then our R = mid-1, since k cannot be be in [mid+1, R] as A[mid+1] > A[mid]

→ if A[mid] < B[mid], Then our L = mid-1, since k cannot be in [L, mid-1] with similar argument as above.

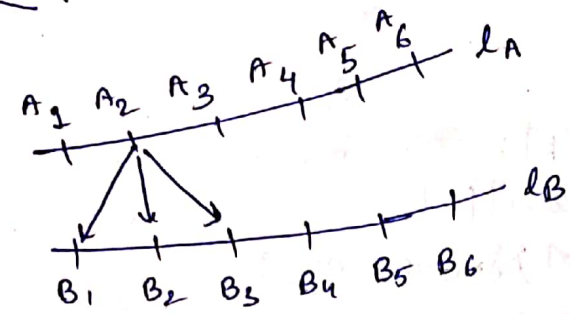
→ Hence, with each iteration the array size reduces and finally we can find 'k' at end of 'i'th iteration.

Hence, P(i) is proved.

Thus, we have established correctness of our algorithm.

Q7 MOKSH SHUKLA 180433

Algorithm :



(1) We need to arrange points on lines A and B, so same as they will present cartesian system to make calculations easy.
So for for this sort A & B on the basis of x-coordinate.
If x-coordinate of all are same, then sort on basis of y-coordinate.

(2) Choose a point on A and calculate distances wrt points on B. While we calculate we observe distance first decreases and then increases as shown in figure
So as distance starts increasing from the point a point, we choose point just before that on B.

(3) Move to next point on A and start calculating distances from the point chosen in previous step as we can say that all points before that will have more distance (as points are sorted)
Repeat the process in (2).

(4) Keep iterating while simultaneously comparing with the global min. distance and changing it if at any step our distance is less than that.
Hence finally we get points with closest distance.

⇒ Time Complexity : Sorting = O(nlogn)
Traversed through B in a single run and also traversed through A in a single run, because soon. while we traverse A, we find a point in B, then in next iteration start from the last chosen point of B. So both arrays traversed once.

Time = O(n)

Total = O(nlogn)

⇒ Space Complexity : O(1) if not consider A,B
O(n) if A,B considered

⇒ Pseudo code :

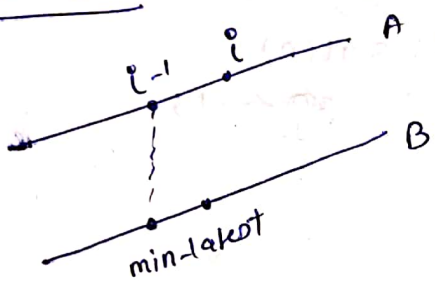
```
struct point {
    int x; int y;
}

Distance-short(A,B) {        → returns 2 arguments
    dist-min ← 1e10;
    min-latest ← 0;
    sort(A); sort(B);
    a1 ← -1; b1 ← -1;
    for (i = 0 to n) {
        j ← min-latest;
        min-temp ← 1e10;
        temp ← Distance(A[i], B[j]);
        while (temp < min-temp) {
            min-latest ← j;
            min-temp ← temp;
            j++;
            if (j == n) break;
            temp ← Distance(A[i], B[j]);
        }
        if (min-temp < dist-min) {
            dist-min = min-temp; }
        a1 ← i; b1 ← min-latest;
    }
    return (A[a1], B[b1])
}
```

⇒ Proof of Correctness :

Assertion P[i] = After i @ iteration we have minimum distanced in dist-min for points b/w (A[0] & A[i]) and (B[0] & B[min-latest])

→ Initially no points so no base case

→ Let P[i-1] holds



→ point nearest to min-latest is i-1.

→ For i to right of i-1, we have already proven that all points to left of min-latest won't be taken into consideration for calculating distance and they will always distance larger than latest min. distance.

→ so we always start from point of min-latest.

→ Now we calculate respective distances for 'i' from min-latest to right.

→ As we have already stated that as distance starts increasing, we stop at that point and hence we have found our shortest distance for i.

Correctness proved

Q8 Algorithm :

(1) Sorted the unsorted array A.
Make a new array B, which contains distinct elements of A, since A has elements which are repeated.
Make another array C, which will contain the frequency of elements of A.
So C[i] = frequency of elements of A present distinctively at B[i]

(2) Define 2 functions lower-bound and upper-bound.

(I) lower-bound (arr, x) : Returns index of first element just greater than equal to x in O(log n) time, in array 'arr'.
It follows divide and conquer approach.

(II) upper-bound (arr, x) : Returns index of first element with value greater than x, in O(log n) time.
Approach is similar to lower-bound.

So, we run these 2 functions in our sorted A with the given query [a,b]. we correspondingly find 2 indices, subtracting which gives no. of elements in A in range [a,b]

(3) Create now a sparse table for frequency array 'C' to answer range-maxima query. (similar to range minima problem in lectures)
In sparse table element at [i] [j] stores index of max occurring element from our array C. (between i and i+j)

(4) Now again using lower-bound and upper-bound, find indice for which B[i] ≥ a & B[i] ≤ b. As we know the range for max query we can easily query our required answer from sparse table already created.

⇛ Pseudo-Code :

```
lower-bound (arr, x) {
        n ← arr.size ;
        L ← 0 ; R ← n ;
        while (L < R) {
            mid = (L+R)/2 ;
            if (arr[mid] >= x) { R ← mid; }
            else { L ← mid+1 ; }
        }
        return R ;
    }
upper-bound (arr, x) {
        n ← arr.size;
        L ← 0 ; R ← n;
        while ( L <= R) {
            mid = (L+R)/2 ;
            if (arr[mid] > x) { R ← mid;}
            else { L ← mid+1 ;}
        }
        return R;
        }

Sparse (A, n) {
        table [n] [log2n + 1] ;
        for (i = 0 to n-1) {
            table [i][0] ← i ;
        }
        for ( j = 1 to log2n) {
            for ( i = 0 to n - 2^j) {
                index 1 ← table [i] [j-1] ;
                index 2 ← table [i + 2^(j-1)] [j-1];
                if ( A[index 1] > A[index 2]) { table [i][j] ← index 1 ;}
                else { table [i][j] ← index 2 ;}
        }}
        return table ; }
```

⇒ Time Complexity :

Preprocessing: Sorting array = $O(n \log n)$

Sparse Table Creation = $O(n \log n)$

overall = $O(n \log n)$

→ lower-bound query = $O(\log n)$
→ upper-bound query = $O(\log n)$
→ Sparse-table query = $O(1)$

overall = $O(\log n)$

⇒ Space Complexity :

Sparse Table = $O(n \log n)$

arrays A, B, C = $O(n)$

overall = $O(n + n \log n) = O(n \log n)$