Todays Content :

- → i^{st} smaller on left → i^{st} smaller on right
- → i^{st} larger on left → i^{st} larger on right
- → Area of Histogram
- → Sum of Max of every subarray

1st smaller element on left side

Given ar[N], for every ele ar[i] find **nereast smaller element** on leftside.

nercarest ele(2) ar[i]
↳ distana between indices

	0	1	2	3	4	5
Ex1: ar[6] =	4	5	2	10	3	12
ans[6] =	-1	4	-1	2	2	3

	0	1	2	3	4	5	6	7
Ex2: ar[8] =	4	6	10	11	7	8	3	5
ans[8] =	-1	4	6	10	6	7	-1	3

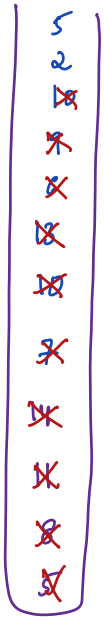
Ideal: for every element, iterate on left & get 1st smaller on left side

```
int[] smallerleft(int ar[n]) { TC: O(N2) SC: O(1)
    int ans[n] = {-1} // if no smaller, it should be -1
    i=0; i<n; i++){
        // find 1st smaller for ar[i] ?
        j=i-1; j>=0; j--){
            // ar[j] is 1st smaller?
            if(ar[j]<ar[i]){
                ans[i] = ar[j];
                break;
            }
        }
    }
}
```

N^2: optimization
↳ NlogN: sorting *
: Binary search *
N times ?
→ target
→ search space
→ discard or not *

Note: doubts/ ideas: public chat

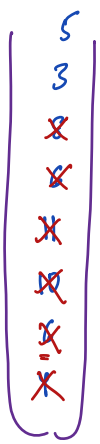
	0	1	2	3	4	5	6	7	8	9	10	11
Eg1: ar[12] =	5	8	11	14	7	10	13	6	9	10	2	5
ans[] =	-1	5	8	11	5	7	10	5	6	9	-1	2



Container: top(), push(): at top pop(): top element

↳ Stack:

	0	1	2	3	4	5	6	7
Eg2: ar[8] =	4	6	10	11	6	8	3	5
ans[8] =	-1	4	6	10	4	6	-1	3



```
int[] 1st smaller(int ar[N]) { TC: 2*n = O(N)  SC: O(N)
    int ans[n] = {-1}
    Stack<int> st;
    i=0; i<n; i++) {
        //ar[i] find 1rr smaller
        while( st.size() > 0 && st.top() >= ar[i] ) {
            st.pop()
        }
        if( st.size() > 0 ) {
            ans[i] = st.top();
        }
        st.push(ar[i])
    }
    return ans;
}
```

top cannot be ans hence we are removing

Nearest smaller index on left

	0	1	2	3	4	5	6	7
ar[8] =	4	6	10	11	7	8	3	5
ans[8] =	-1	0	1	2	.....			

0	1	2	3

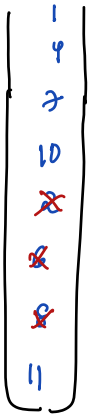
```
int[] 1st smaller Index(int ar[N]) { TC: O(N)  SC: O(N)
    int ans[n] = {-1}
    Stack<int> st;
    i=0; i<n; i++) {
        //ar[i] find 1rr smaller
        while( st.size() > 0 && ar[st.top()] >= ar[i] ) {
            st.pop()
        }
        if( st.size() > 0 ) {
            ans[i] = st.top();
        }
        st.push(i) // only push index
    }
} return ans;
```

TODO: 1st small index on right:

Iterate from n-1 → 0

2) Nearest Greater on left side

	0	1	2	3	4	5	6	7
Eg1: ar[8] =	11	8	6	2	10	7	4	1
ans[] =	-1	11	8	6	11	10	7	4




```
int[] 1st greater left(int ar[N]) {
    int ans[n] = {-1}
    stack<int> st;
    i=0; i<n; i++) {
        //ar[i] find 1st smaller
        while( st.size() >0 && st.top() >= ar[i]) {
            st.pop()
        }
        if( st.size() >0 ){
            ans[i] = st.top();
        }
        st.push (ar[i])
    }
    return ans;
}
```

top cannot be ans hence we are removing

```
ar[st.top()] >= ar[i]
//1st greater index left
st.push(i)
```

Note: 1st greater index on right:

Iterate from right to left

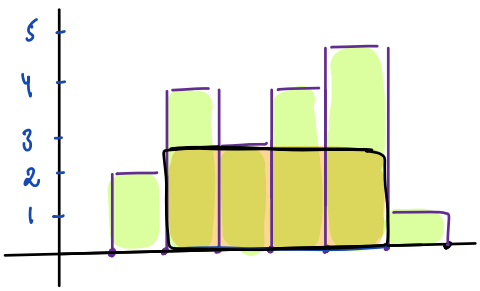
10:16 → 10:25 pm

Histogram area:

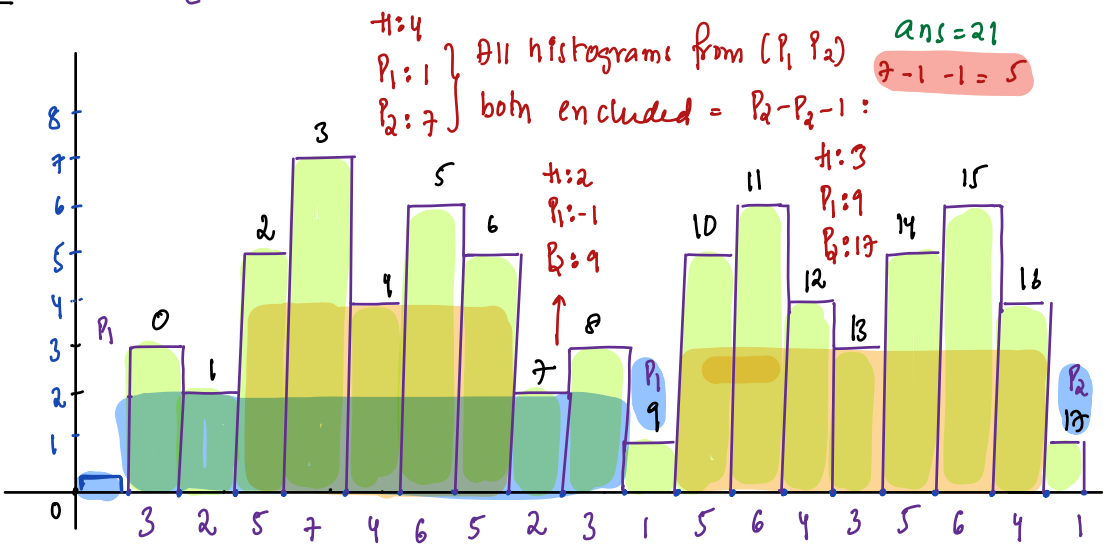
Given Continous block of histogram find max Rectangular area which can be present with in histograms

Note: Every histogram is of width = 1

	0	1	2	3	4	5	
Eg1: ar[6] = {	2	4	3	4	5	1 }	area = 12



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Eg2: ar[] = {	3	2	5	7	4	6	5	2	3	1	5	6	4	3	5	6	4	1 }



obs1: Height of a rectangle should be one of the height of histogram

Ideal: for every histogram height take it as rectangle height

: Iterate on right & get 1st small index whose value is < rectangle height = P2

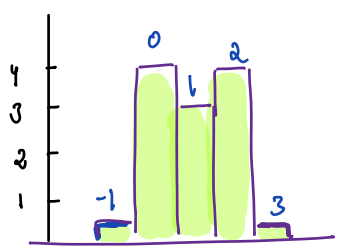
: Iterate in left & get 1st small index whose value is < rectangle height = P1

: area = max(ans, (P2 - P1 - 1) * height) // current rectangular height

```
int RectArea(int hist[n]){   TC: O(N+N+N) = O(N)   SC: O(n+n) = O(N)
    int left[] = 1st small inden left(hist[7])
                 // if no smaller on left initialize with -1   } Handle Edge Case
    int right[] = 1st small inden right(hist[])
                 // if no smaller in right initialize with N
    area = 0
    for(i=0; i<n; i++){ // consider every histogram rectangle height
        int h = hist[i] // height of histogram
        // 1st small inden on left & 1st small inden in right
        int P1 = left[i]        int P2 = right[i]
        area = max(area, [P2-P1-1]*h)
    }
    return area;
}
```

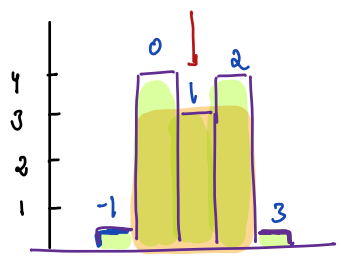
Edge Case:

arr[3] = 4 3 4 (indices 0 1 2)



P1: left :	-1	-1	1
P2: right :	1	-1	-1
{P2 - P1 - 1}			
width :	1	-1	-3

{It cannot be negative}



P1: left :	-1	-1	1
P2: right :	1	3	3
width :	1	-3	1

3 - (-1) - 1
3 + 1 - 1 = 3

3Q) Given ar[N] distinct elements, sum of max of every subarray

Eg: ar[3] = { 4 2 3 } ans = 20

	max
{4}	4
{4 2}	4
{4 2 3}	4
{2}	2
{2 3}	3
{3}	3

4*3 + 3*2 + 2*1 = 20

Idea: Contribution technique

- for every element, calulate in how many subarrays, ar[i] is max, say its c_i
- Contribution = ar[i] * c_i

```
ans = 0
for(i = 0; i<n; i++){
    ans = ans + ar[i] * c_i
}
```

c_i : no: of subarrays in which ar[i] is max

Eg1: ar[] = { 1 3 2 10 1 4 2 8 6 4 14 2 19 }

0	1	2	3	4	5	6	7	8	9	10	11
1	3	2	10	1	4	2	8	6	4	14	2	19

P_1 [$s = i - P_1$] i [$e = P_2 - i$] P_2

s: 3, 4, 5, 6 — e: 6, 7, 8

count of subarrays = 12

$s = i - P_1$ # $e = P_2 - i$ # $c_i = s * e = (i - P_1) * (P_2 - i)$

P_1 = i^{st} greater index in left
Initialize with -1

P_2 = i^{rr} greater index in right
Initialize with N

```
int manSub(int ar[N]){ TC: O(N) SC: O(N)
    TODO
}
```

→ Sum of min of every subarray : TODO ✓

→ If elements are not distinct: {Edge Case}

→ Sum of max of every subarray : TODO ✓

→ Sum of min of every subarray : TODO ✓

→ golang:

- → struct
- → pointers
- → pointer as member in struct
- → linked list in go-lang

Doubts: