

Stacks 2: Nearest Smaller/Greater Element

1.Problem Statement –nearest smaller element on left

Given an array. Find the index of nearest smallest element on left for all i index in $A[]$. For all i . find j , such that $A[j] < A[i]$, $j < i$ and j is maximum. Given an array A , find the nearest smaller element $G[i]$ for every element $A[i]$ in the array such that the element has an index smaller than i . More formally,

$G[i]$ for an element $A[i]$ = an element $A[j]$ such that

j is maximum possible AND

$j < i$ AND

$A[j] < A[i]$

Elements for which no smaller element exist, consider the next smaller element as -1 .

Problem Constraints

$1 \leq |A| \leq 100000$

$-10^9 \leq A[i] \leq 10^9$

Input Format

The only argument given is integer array A .

Output Format

Return the integer array G such that $G[i]$ contains the nearest smaller number than $A[i]$.

If no such element occurs $G[i]$ should be -1 .

Example Input

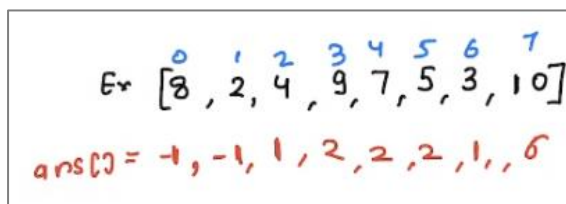
Input 1:

$A = [4, 5, 2, 10, 8]$

Example Output

Output 1:

$[-1, 4, -1, 2, 2]$



Here last index is checked first. Whichever we push in the end, we access it first \rightarrow this is stack structure \rightarrow LIFO

Brute Force

for each index i {

 iterate j from $i-1$ to 0 until you find the smaller element{

 update $ans[i]$

 }

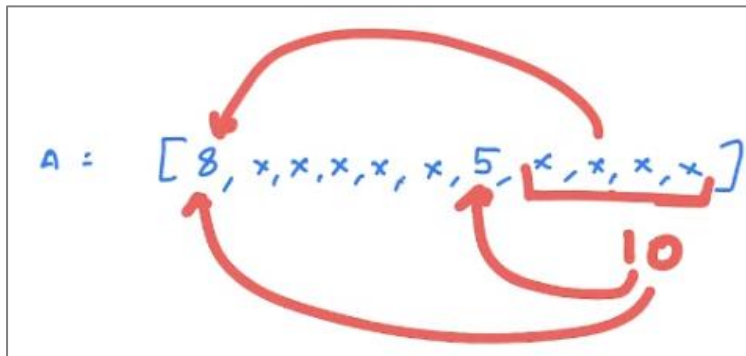
}

TC = $O(N^2)$

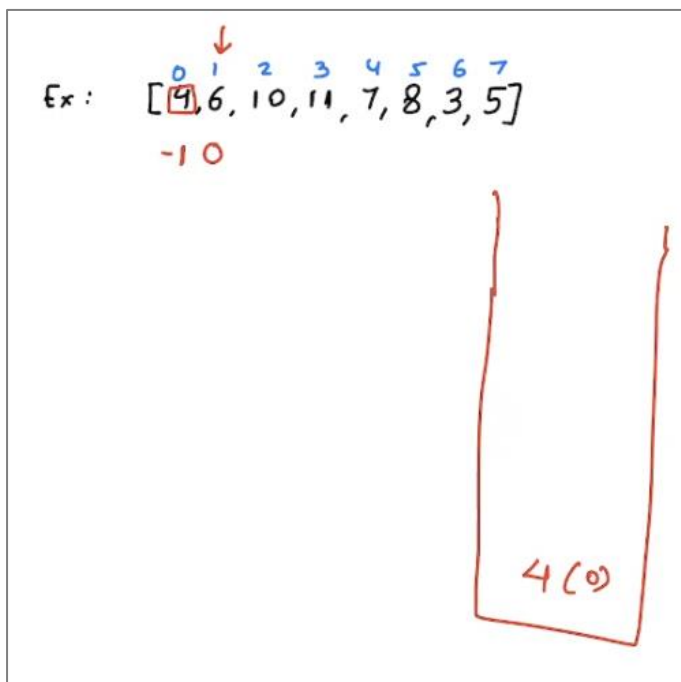
SC = $O(1)$

Idea -2

Observation:



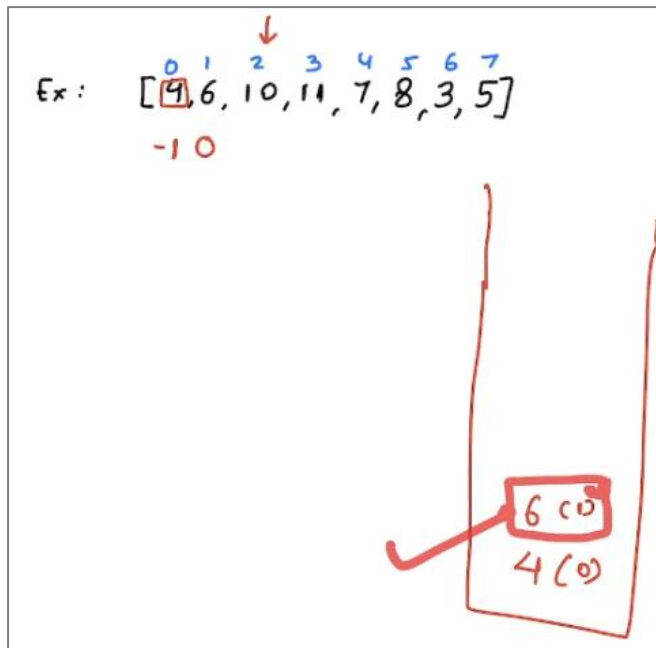
For any element after 5, could 8 become nearest smallest element \rightarrow No because 5 is smaller than 8.



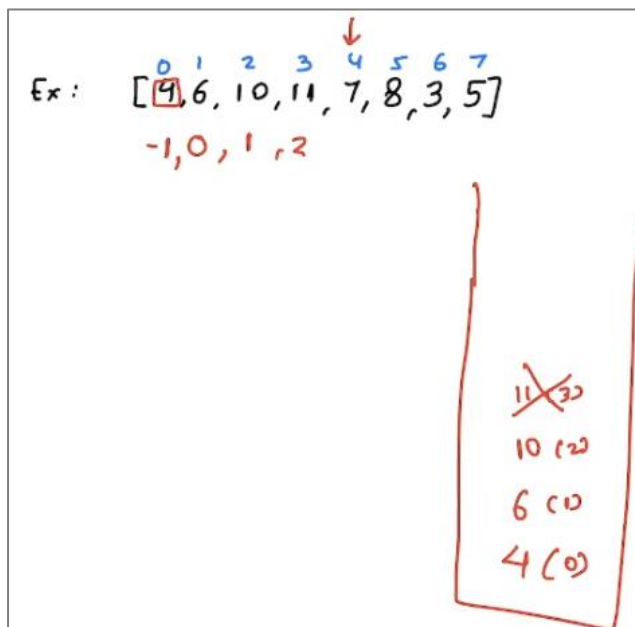
For element 4, do we have any possible candidate \rightarrow No \rightarrow so put -1. But it be possible that 4 can be nearest smaller element for any other number

For element 6, could it be possible that 4 can be nearest smaller element for any other number → Yes. If 6 would have been smaller than 4, then we could say that 4 will not become nearest smaller element for numbers on the right side.

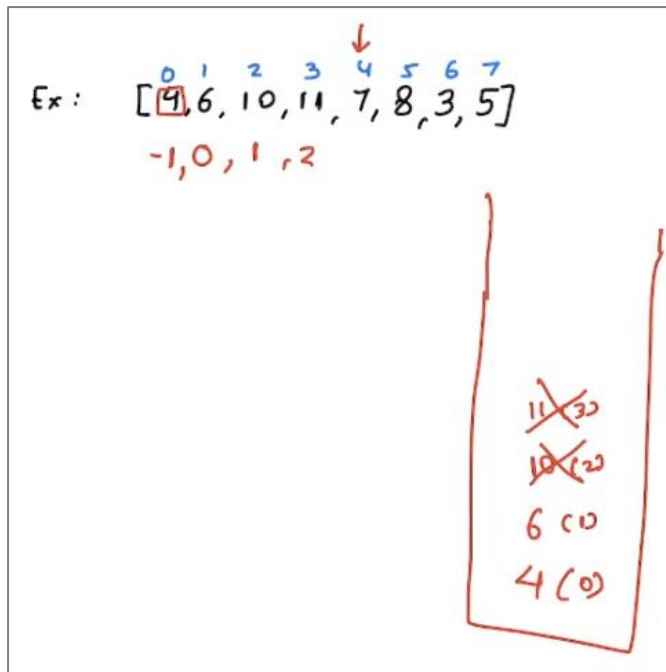
But 6 can also be the possible candidate for number greater than 6. So adding 6 in the possible answer



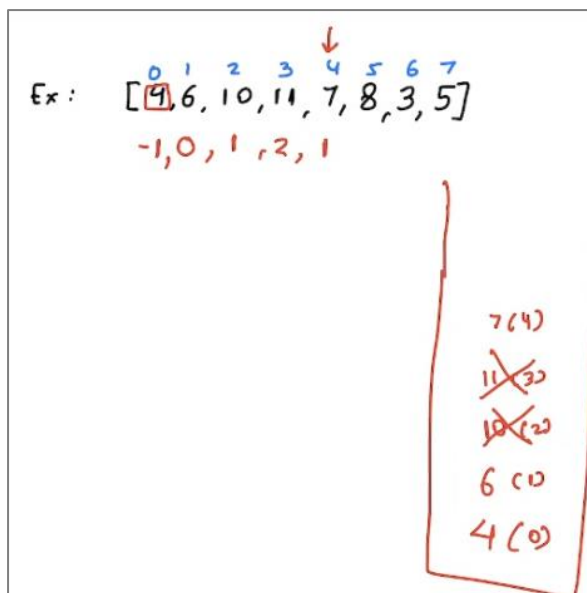
For element 7, now that 7 is already there 11 cannot become nearest smaller



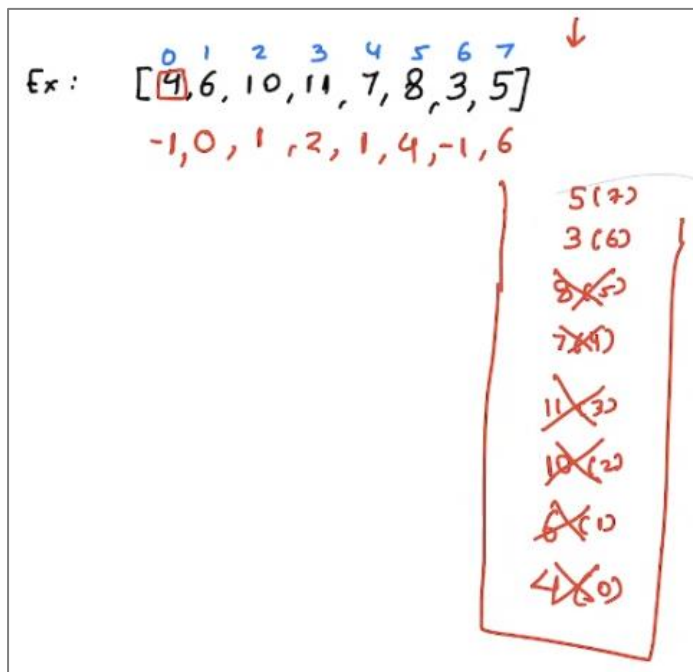
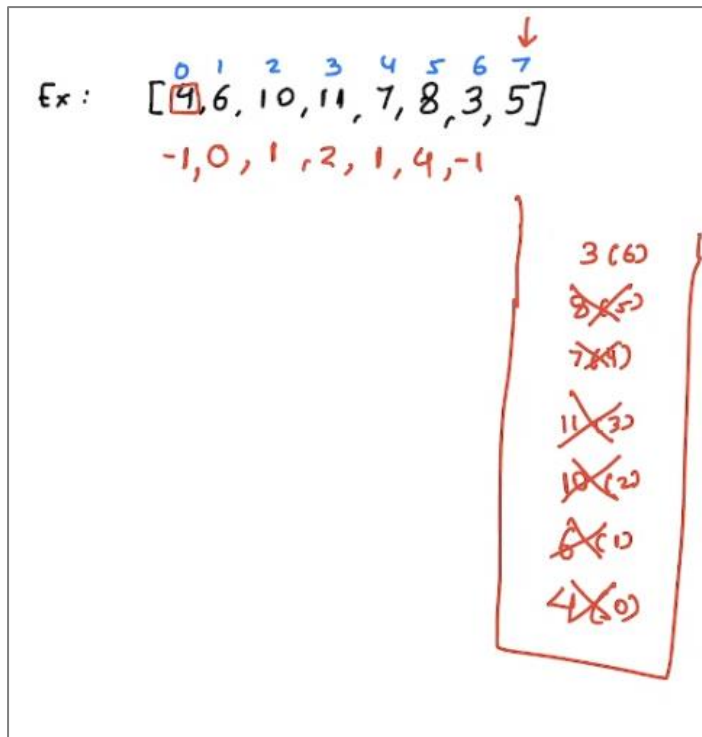
Same with 10. so remove 10 also



But $6 < 7$, so we cannot remove 6 but we add element 7.



For 3 element, $8 > 3$, remove 8 and keep deleting until number < 3



This is the answer. In my stack, I will add indexes and not element because, It is possible to get value of $A[i]$ with the index but when we store $A[i]$ as a number, we cannot know the index and in the question, 'index' as ans is asked.

for each index i

Keep deleting elements from top
until they are $\geq A[i]$

if (st is not empty) $ans[i] = top$
else $ans[i] = -1$

add i in stack as possible answer.

Stack <int> st

for ($j=0$; $j<N$; $j++$)

while ($st.size > 0$ & $A[st.top()] \geq A[i]$)
{
 $st.pop()$

if ($st.empty()$) $ans[i] = -1$
else $ans[i] = st.top()$

$st.push(i)$

return ans

Here with $st.top()$ \rightarrow we are storing the indexes

```

for (i=0 to n-1) {
    while(!st.isEmpty() && A[st.peek()] >= A[i]) {
        st.pop()
    }
    if (st.isEmpty()) ans[i] = -1
    else ans[i] = st.peek()
    st.push(i)
}

return ans

```

$TC = \cancel{O(N^2)} \rightarrow O(N)$
 $SC = O(N)$

total N insertions in stack
 \Downarrow
 total only N deletions

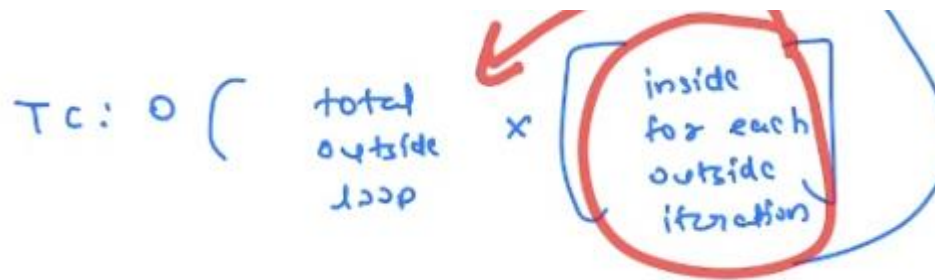
\Rightarrow while loop will
run N times in total

while loop will only remove an element. Insertions are done at the end. And there are only N insertions. If there N insertions, there will be N deletions.

We cannot delete the stack if it is already empty so while loop will run N times in total

TC: $O \left(\begin{array}{c} \text{total} \\ \text{outside} \\ \text{loop} \end{array} + \begin{array}{c} \text{total} \\ \text{inside} \\ \text{loop} \end{array} \right)$

For each outside, how many inside loop is running then we will do “*”. When I am saying total → then I will add



$$Tc: O \left(\begin{array}{c} \text{total} \\ \text{outside} \\ \text{loop} \end{array} + \begin{array}{c} \text{total} \\ \text{inside} \\ \text{loop} \end{array} \right) : O(N) : O(N)$$

N + N

SC = O(N)

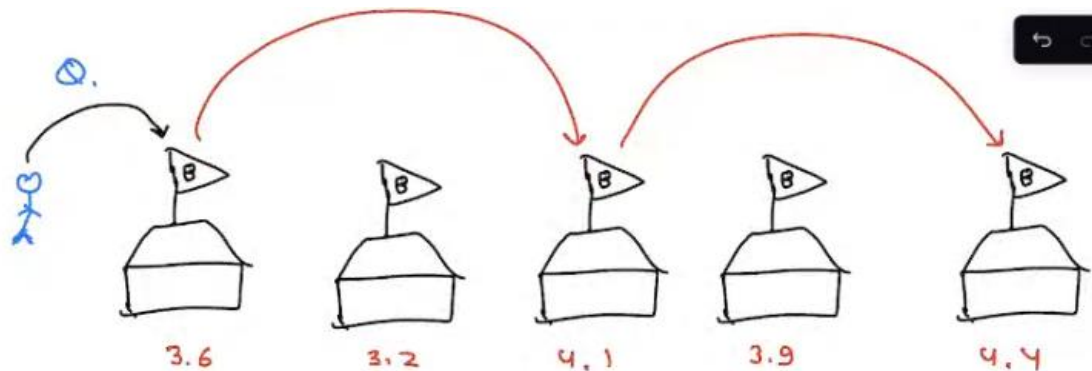
Here inside while loop, pop() will run only one time.

Actual Code

```
public class Solution {
    public ArrayList<Integer> prevSmaller(ArrayList<Integer> A) {
        Stack<Integer> st = new Stack<>();
        ArrayList<Integer> ans = new ArrayList<>();
        for(int i=0; i<A.size(); i++){
            Integer ch = A.get(i);
            while(!st.isEmpty() && st.peek() >= ch){
                st.pop();
            }
            if(st.isEmpty()){
                ans.add(-1);
            }else{
                ans.add(st.peek());
            }
            st.push(ch);
        }
        return ans;
    }
}
```


2.Problem Statement – Nearest larger element on right side

There are couple of restaurants and all of them have google maps ratings. First you visit the restaurant with 3.6 ratings and turn out this restaurant doesnot meet your expectations. Now your target would be nearest restaurants with >3.6 rating in the right side



Find the nearest larger element on the right side. Example:

0	1	2	3	4	5	6
3	2	6	5	8	7	9
2	2	4	4	6	6	-1

```

Stack <int> st
for ( j = N-1 ; j >= 0 ; j-- )
{
    while ( st.size > 0 & A[st.top()] <= A[j] )
    {
        st.pop()
    }

    if ( st.empty() ) ans[j] = -1
    else ans[j] = st.top()

    st.push(j)
}

return ans

```

Two changes → direction changes from N-1 and <= sign change

In an arrangement of 6, 4. For element 6, 4 will not be possible ans because it is smaller than 6 and hence won't possible ans on any number on the left of 6.

```

for (i=0 to n-1) {
    while(!st.isEmpty() && A[st.peek()] >= A[i]) {
        st.pop();
    }
    if (st.isEmpty()) ans[i] = -1;
    else ans[i] = st.peek();
    st.push(i);
}

return ans;

```

Q2 → i , find the nearest smaller or equal element on left
 Q3 → i , find the nearest greater element on left
 Q4 → i , find the nearest greater or equal element on left
 Q5 → i , find the nearest smaller element on right

2. Problem Statement – Largest Rectangle in histogram

Given an array A, where $A[i]$ = height of the i th bar [width of all bar is 1]

Find the area of the largest rectangle formed by continuous bars

Given an array of integers A. A represents a histogram i.e $A[i]$ denotes the height of the i th histogram's bar. Width of each bar is 1. Find the area of the largest rectangle formed by the histogram.

Problem Constraints

$1 \leq |A| \leq 100000$

$1 \leq A[i] \leq 10000$

Input Format

The only argument given is the integer array A.

Output Format

Return the area of the largest rectangle in the histogram.

Example Input

Input 1:

A = [2, 1, 5, 6, 2, 3]

Input 2:

A = [2]

Example Output

Output 1:

10

Output 2:

2

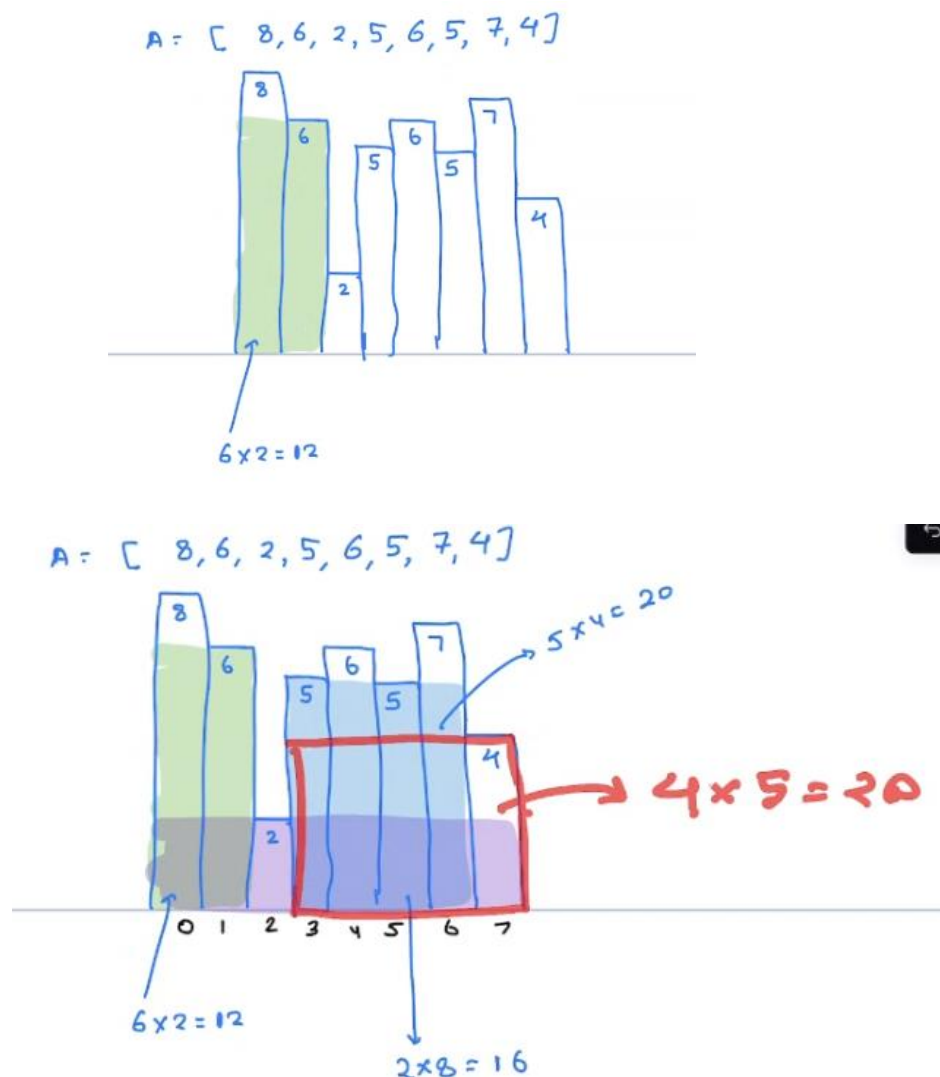
Example Explanation

Explanation 1:

The largest rectangle has area = 10 unit. Formed by A[3] to A[4].

Explanation 2:


Largest rectangle has area 2.



Brute Force – for all subarray (i, j), find height & width and calculate area. Max area is the answer. With carry forward

```
ans = 0
for (i = 0 to n-1) {
    H = a[i]
    for (j = i to n-1) {
        H = min(H, a[j])
        W = j - i + 1
        ans = max(ans, H * W)
    }
}
return ans
```

BF for each subarray



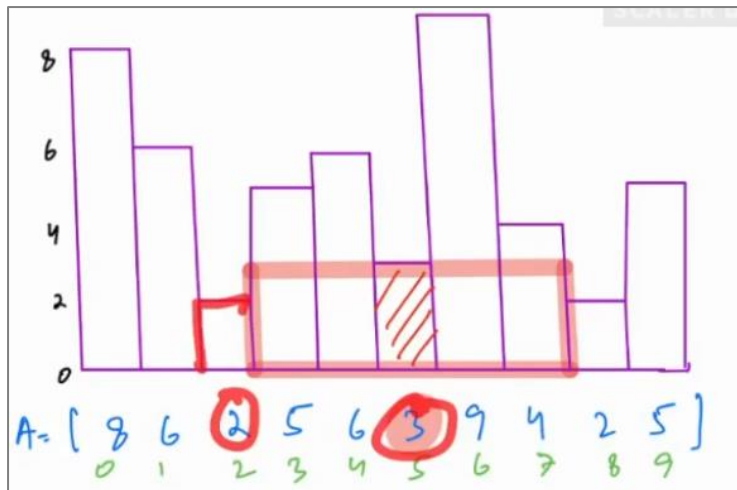
1. min of the subarray
2. ans_temp = min x len of subarray
3. update ans

return ans

TC = $O(N^2)$

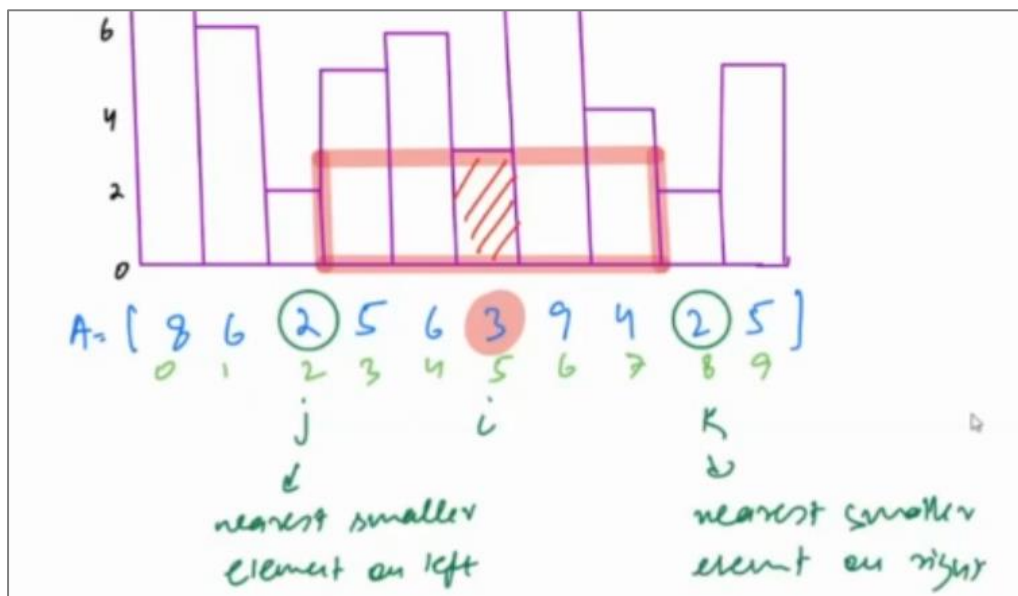
SC = $O(1)$

Optimized Solution:



For element 3, 2 is the nearest smaller element in the left and another 2 is nearest smaller element in the right.

Both 2 will break the chain and we will be left with this highlighted rectangle.



So for each element we need to find nearest smaller element on left and right. For this rectangle, height = $A[i]$ and width for subarray $[j+1, k-1]$

width = $k-1-(j+1) + 1 = k-j-1$ since length = $b-a+1$



One end of the rectangle will be $j+1$ and $k+1$

```

3.  ans = 0
    for (i=0 to n-1) {
        j = left[i] → if (j == -1) j+1 = 0 which is OK
        k = right[i] → if (k == -1) k-1 = -2 which is not OK
        if (k == -1) k = n → k+1 = n-1 which is OK
        ans = max(ans, a[i] * (k-j-1))
    }
    return ans

```

$j = -1 \quad k = n$
 $w = n - (-1) - 1$
 $= n + 1 - 1 = n$

If there is no element in the right, last index = -1, If $k == -1$, $k = n$ which is the total length of the array. If $j = -1$ and $k = n$, then width = $n - (-1) = n$ which is the whole width of the array.

SC = $O(N)$

total TC = $O(N + N + N)$

\uparrow \uparrow \uparrow
 create create main
 left() right() logic
 array array

for each i

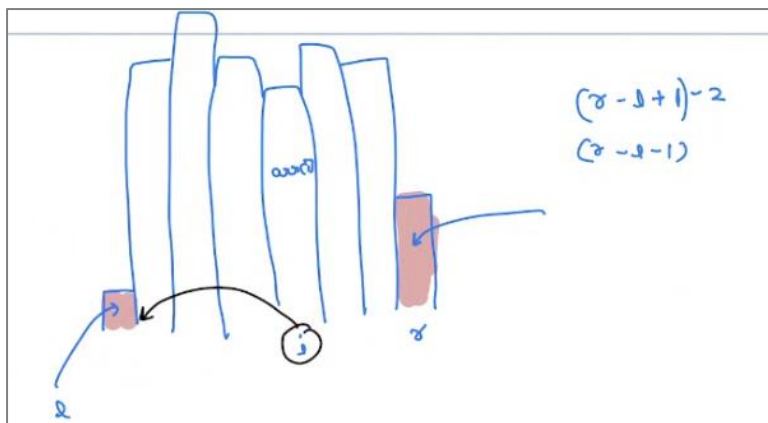
consider $arr[i]$ is the height
 $r =$ nearest element in right which is smaller
 $l =$ nearest element in left which is smaller
 $ans_tmp = \frac{arr[i] \times (r - l - 1)}{2}$
 update ans
 return ans

Default value of $r = N$

For each $A[i]$, find the index of the element which is smaller on the right side let's name as r ,

similarly for element which is smaller on the left side let's say l

Also consider the last index and first index if we keep looking for smaller element in the right or left side.



we cannot include r & l in the code since they are smaller. so base length = $r-l+1-2 = r-l-1$
 Area = height * base length

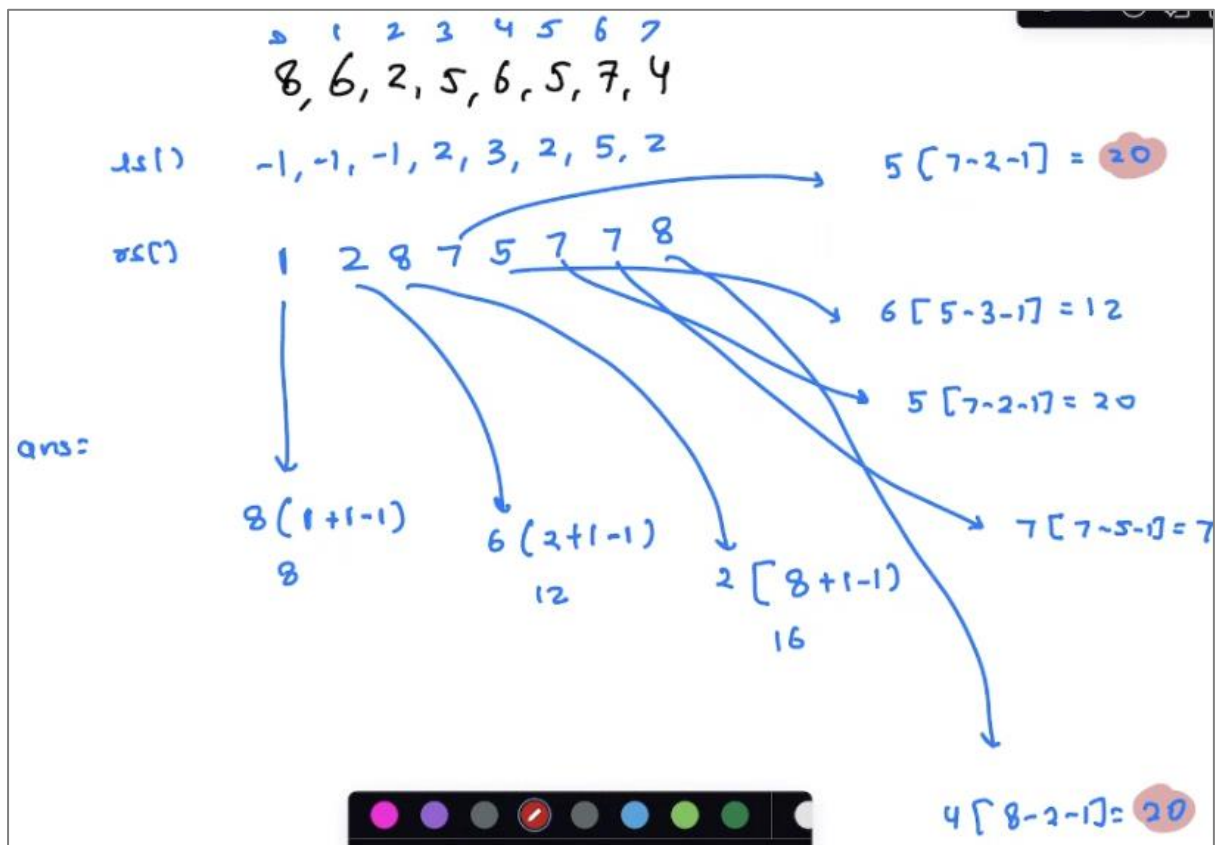
```

for each i
{
    consider arr[i] is the height
    r = rsc[i]
    l = lsc[i]
    ans_tmp = arr[i] * (r - l - 1)
    ans = max(ans, ans_tmp)
}
return ans
    
```

TC = $O(N+N+N) = O(N)$

SC = $O(N+N) = O(N)$ we used two stacks

Dry run:



Actual Code

```

public class Solution {
    public int largestRectangleArea(ArrayList<Integer> A) {
        int n = A.size();
        int[] left = new int[n];
        int[] right = new int[n];

        Stack<Integer> st = new Stack<>();
        for(int i =0; i<n; i++){
            while(!st.isEmpty() && A.get(st.peek()) >= A.get(i)){
                st.pop();
            }
            left[i] = st.isEmpty()? -1: st.peek();
            st.push(i);
        }
        st.clear();

        for(int i = n-1; i>=0; i--){
            while(!st.isEmpty() && A.get(st.peek()) >= A.get(i)){
                st.pop();
            }
            right[i] = st.isEmpty()? n: st.peek();
            st.push(i);
        }
        int maxArea =0;
        for(int i =0; i<n; i++){
            int width = right[i] - left[i] -1;
            maxArea = Math.max(maxArea, A.get(i) * width);
        }
        return maxArea;
    }
}

```

3.Problem Statement – Find max-min

Given an integer array of distinct elements. Find (max-min) for all subarrays and return their sum as answer.

Ex: [2,5,3]

Ex (2, 5, 3)

	max	min	max-min
2	2	2	0
5	5	5	0
3	3	3	0
2 5	5	2	3
5 3	5	3	2
2 5 3	5	2	3

$3 + 2 + 3 = 8$
 ans = 8

Brute Force:

Carry Forward → For each subarray, Calculate max and min separately → then do max - min

```

ans = 0
for (i = 0 to n-1) {
  mx = a[i], mi = a[i]
  for (j = i to n-1) {
    mx = max(mx, a[j])
    mi = min(mi, a[j])
    ans += (mx - mi)
  }
}
return ans
  
```

$TC = O(N^2)$
 $SC = O(1)$

$i=0$ $i=1$
 $j=0, 1, 2$ $j=1, 2$

Calculate summation of all max of each subarray and summation of all min of each subarray separately and then do max-min

Optimize using Contribution Technique

For all i, contribution of a[i]

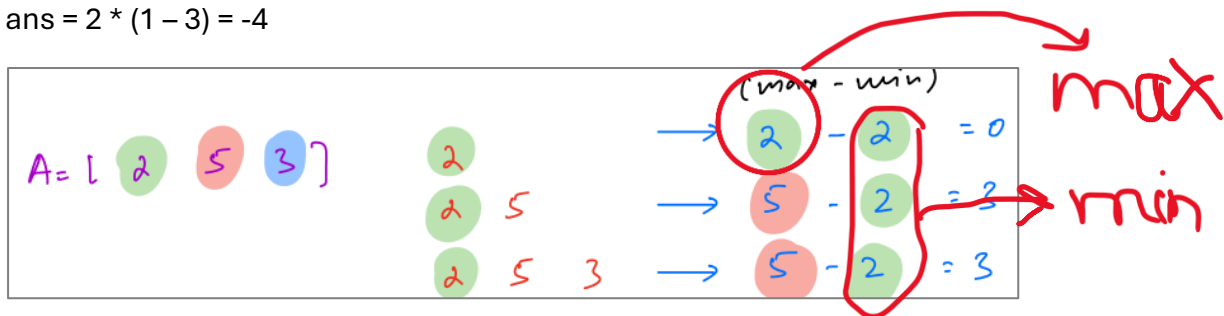
$$ans = \sum_i \text{contribution of } a[i]$$

↓

$$a[i] * \left(\begin{array}{l} \# \text{ subarrays where} \\ a[i] \text{ is max} \end{array} - \begin{array}{l} \# \text{ subarrays} \\ \text{where } a[i] \text{ is} \\ \text{min} \end{array} \right)$$

For element 2, 2 is max in some subarray and min in other.

$$ans = 2 * (1 - 3) = -4$$



$$ans = \sum (max - min)$$

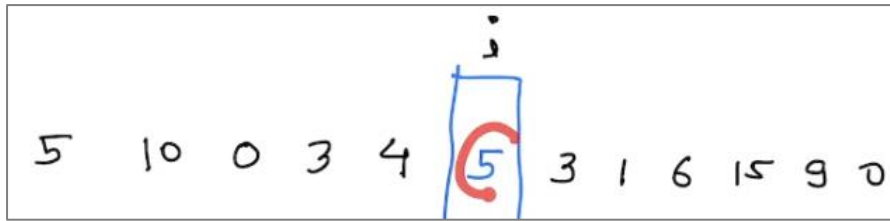
ans = $\sum \text{max of each subarray}$ \rightarrow $\sum \text{min of each subarray}$

$$\sum \text{max of each subarray} = \begin{array}{l} arr[0] \\ \times \\ \text{No of subarray} \\ \text{arr[0] is max} \end{array} + \begin{array}{l} arr[1] \\ \times \\ \text{No of subarray} \\ \text{arr[1] is max} \end{array}$$

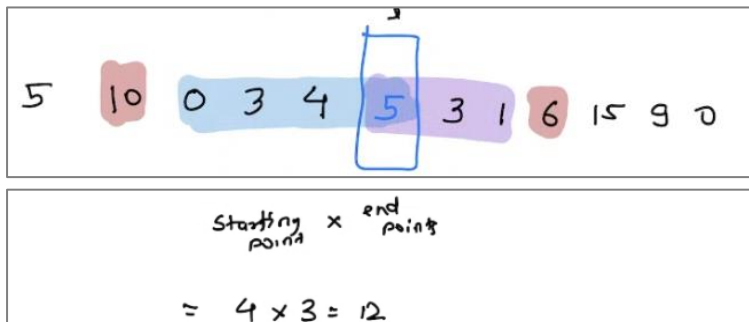
$$+ \begin{array}{l} arr[2] \\ \times \\ \text{No of subarray} \\ \text{arr[2] is max} \end{array} + \dots$$

How do I know in how many subarrays $arr[0]$ is maximum.

EX:



In the above example If I want 5 to be max, if in my subarray if 3 or 4 or 0 is there my 5 is max but if 10 or 6 is there then 5 will not be max in my subarray. Element 5 will be greater until you find nearest element greater than 6 in the left and in the right.



In this array, we can say that there are 12 subarrays in which 5 is max. starting point is from 0, 3, 4 --- and ending point is ...3,1. Subarrays can start from 0 and end at 1.

0 3 4 5

3 4 5

4 5

and so on..

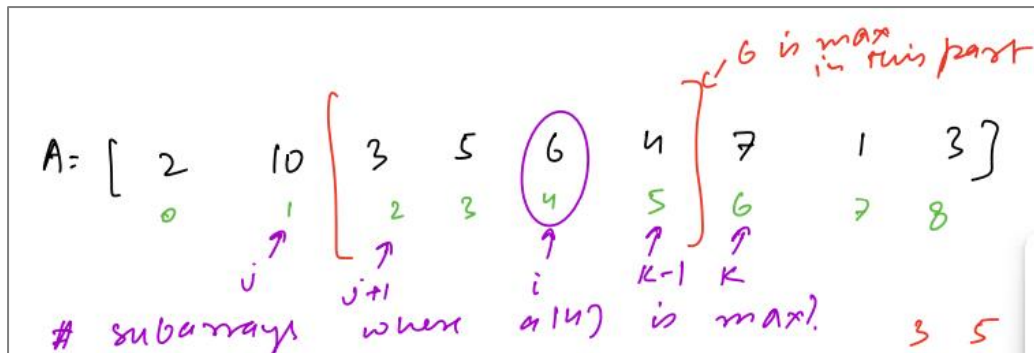
So Start index of a subarray should be less than or equal to i and index should be greater than index of nearest greater element on left.

$$\left. \begin{array}{l} \text{nearest greater} < \text{start index} \leq i \\ \text{on left (j)} \end{array} \right\} [j+1, i] \rightarrow \text{count} = (i-j)$$

If index of nearest greater element on left is j , the start and end index of subarray should be $j+1, i$ and length would be $i - j + 1 - 1 = i - j$ where i = element 5

End index would be

$$\left. \begin{array}{l} i \leq \text{end index} < \text{nearest greater} \\ \text{on right (k)} \end{array} \right\} [i, k-1] \rightarrow \text{count} = (k-i)$$



of subarrays calculated using contribution technique

$$\text{\# subarrays where } a[i] \text{ is max} = (i - j) \times (k - i)$$

$$\text{\# subarrays where } a[i] \text{ is max} = (i - \text{greaterLeft}[i]) \times (\text{greaterRight}[i] - i)$$

Handwritten note: "nearest greater element in left" with an arrow pointing to $\text{greaterLeft}[i]$.

$\text{greaterLeft}[i]$ is j

$\text{greaterRight}[i]$ is k

$$\text{\# subarrays where } a[i] \text{ is min} = (i - \text{smallerLeft}[i]) \times (\text{smallerRight}[i] - i)$$

dry run

Array $A = [2, 5, 3]$

Indices: $0, 1, 2$

$\text{greaterLeft} = [-1, -1, 1]$

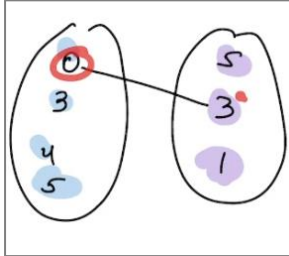
$\text{greaterRight} = [1, 3, -1]$ (for -1 replace with n)

$\text{smallerLeft} = [-1, 0, 0]$

$\text{smallerRight} = [3, 2, 3]$

contribution of $a[i]$

$$= a[i] * \left(\begin{matrix} \# \text{ subarrays where} \\ a[i] \text{ is max} \end{matrix} - \begin{matrix} \# \text{ subarrays} \\ \text{where } a[i] \text{ is} \\ \text{min} \end{matrix} \right)$$



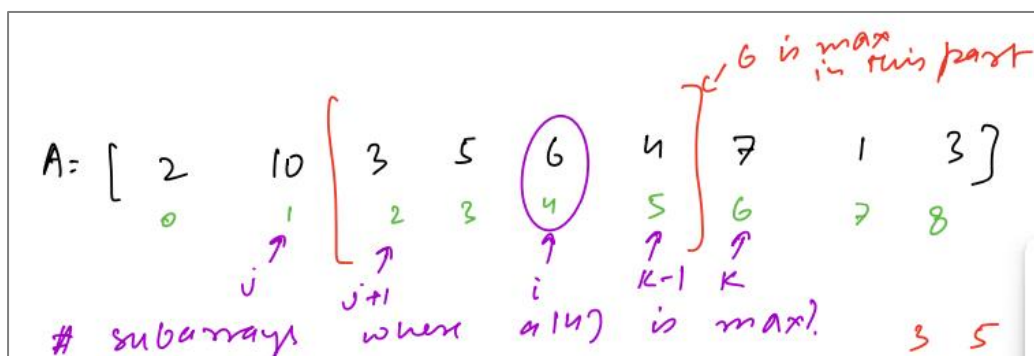
For i , find l & r .

of starting points are $= i - l + 1 - 1 = i - l$

of ending points $= r - l + 1 - 1 = r - l$

No. of

$$\text{No. of subarrays} = (i - l)(r - l)$$



Actual Code

```
public class Solution {
    public int solve(ArrayList<Integer> A) {
        int n = A.size();
        long summax = 0, summin = 0;
        long mod = 1000000007L;
        ArrayList<Integer> GL = new ArrayList<>(Collections.nCopies(n, -1));
        ArrayList<Integer> GR = new ArrayList<>(Collections.nCopies(n, n));
        ArrayList<Integer> SL = new ArrayList<>(Collections.nCopies(n, -1));
        ArrayList<Integer> SR = new ArrayList<>(Collections.nCopies(n, n));

        greaterLeft(A, GL);
        greaterRight(A, GR);
        smallerLeft(A, SL);
    }
}
```

```

smallerRight(A, SR);

for(int i=0; i < A.size(); i++){
    long leftmax = i - GL.get(i); //j+1 to i; i-(j+1)+1//Gives the Number of elements to the left where A[i] is maximum
    long rightmax = GR.get(i) - i; //i to k-1; k-1-i+1// Gives Number of elements to the right where A[i] is maximum
    //if (rightmax == -1) rightmax = A.size();
    long maxcontri = (A.get(i) % mod * leftmax % mod * rightmax % mod) % mod;
    summax = ((summax) % mod + (maxcontri) % mod) % mod;

    long leftmin = i - SL.get(i);
    long rightmin = SR.get(i) - i;
    //if (rightmin == -1) rightmin = A.size();
    long mincontri = (A.get(i) % mod * leftmin % mod * rightmin % mod) % mod;
    summin = ((summin) % mod + (mincontri) % mod) % mod;
}
return (int)((summax - summin + mod) % mod);
}

private void greaterLeft(ArrayList<Integer> A, ArrayList<Integer> GL){
    Stack<Integer> st = new Stack<>();
    for(int i = 0; i<A.size(); i++){
        while(!st.isEmpty() && A.get(st.peek()) < A.get(i)){
            st.pop();
        }
        if(!st.isEmpty()){
            GL.set(i, st.peek());
        }
        st.push(i);
    }
}

private void greaterRight(ArrayList<Integer> A, ArrayList<Integer> GR){
    Stack<Integer> st = new Stack<>();
    for(int i = A.size()-1; i>=0; i--){
        while(!st.isEmpty() && A.get(st.peek()) <= A.get(i)){
            st.pop();
        }
        if(!st.isEmpty()){
            GR.set(i, st.peek());
        }
        st.push(i);
    }
}

private void smallerLeft(ArrayList<Integer> A, ArrayList<Integer> SL){
    Stack<Integer> st = new Stack<>();
    for(int i = 0; i<A.size(); i++){
        while(!st.isEmpty() && A.get(st.peek()) > A.get(i)){

```



```

        st.pop();
    }
    if(!st.isEmpty()){
        SL.set(i,st.peak());
    }
    st.push(i);
}

}
private void smallerRight(ArrayList<Integer> A, ArrayList<Integer> SR){
    Stack <Integer> st = new Stack<>();
    for(int i = A.size()-1; i>=0; i--){
        while(!st.isEmpty() && A.get(st.peak()) >= A.get(i)){

            st.pop();
        }
        if(!st.isEmpty()){
            SR.set(i, st.peak());
        }
        st.push(i);
    }

}

}

```

TC: $O(N)$: $O(N)$
 SC: $O(N)$: $O(N)$

4.Problem Statement- Max Rectangle in Binary Matrix

Given a 2-D binary matrix **A** of size **N x M** filled with **0's** and **1's**, find the **largest** rectangle containing **only ones** and return its **area**.

Problem Constraints: $1 \leq N, M \leq 100$

Input Format: The first argument is a 2-D binary array A.

Output Format: Return an integer denoting the **area** of the largest rectangle containing **only ones**.

Example Input 1:

```

A = [
    [1, 1, 1]
    [0, 1, 1]
    [1, 0, 0]
]

```

Example Output 1: 4

Example Explanation1: As the max area rectangle is created by the 2x2 rectangle created by (0, 1), (0, 2), (1, 1) and (1, 2).

Example Input 2:

```
A = [
    [0, 1, 0]
    [1, 1, 1]
]
```

Example Output 2: 3

Example Explanation2: As the max area rectangle is created by the 1x3 rectangle created by (1, 0), (1, 1) and (1, 2).

Actual Code

```
public class Solution {
    public int maximalRectangle(ArrayList<ArrayList<Integer>> A) {
        if (A == null || A.size() == 0) return 0;
        int n = A.size();
        int m = A.get(0).size();
        int maxArea = 0;

        int [] heights = new int [m];
        for(int i = 0; i < n; i++){
            for(int j = 0; j < m; j++){
                if(A.get(i).get(j) == 1){
                    heights[j] += 1;
                }else{
                    heights[j] = 0;
                }
            }
            maxArea = Math.max(maxArea, largestRectangleArea(heights));
        }
        return maxArea;
    }
    private int largestRectangleArea(int [] heights){
        Stack<Integer> st = new Stack<>();
        int maxArea = 0;
        int n = heights.length;

        for(int i = 0; i <= n; i++){
            int currHeight = (i == n) ? 0 : heights[i];
            while(!st.isEmpty() && currHeight < heights[st.peek()]){
                int height = heights[st.pop()];
                int width = st.isEmpty() ? i : i - st.peek() - 1; // *width → when i = 3, height = heights[2], 3-1-1=1
                // this while will repeat until stack becomes empty
                maxArea = Math.max(maxArea, height * width);
            }
            st.push(i);
        }
    }
}
```

```

    }
    st.push(i);
}
return maxArea;
}
}

```

* width = **Stack is Not Empty** ($i - \text{st.peek()} - 1$)

- If the stack is not empty after popping, the **top of the stack** gives the index of the nearest smaller bar to the left of the popped bar.
- The rectangle's width is the difference between the current index i and the index of the next smaller bar st.peek() , minus 1 (to exclude the smaller bar itself).

5.Problem Statement- Next Greater on right

Given an array **A**, find the **next greater element G[i]** for every element **A[i]** in the array. The next greater element for an element **A[i]** is the first greater element on the right side of **A[i]** in the array, **A**.

More formally:

$G[i]$ for an element $A[i]$ = an element $A[j]$ such that

j is minimum possible AND

$j > i$ AND

$A[j] > A[i]$

Elements for which no greater element exists, consider the next greater element as -

Problem Constraints $1 \leq |A| \leq 10^5$

$1 \leq A[i] \leq 10^7$

Input Format: The first and the only argument of input contains the integer array, **A**.

Output Format: Return an integer array representing the next greater element for each index in **A**.

Example Input1: $A = [4, 5, 2, 10]$

Example Output1: $[5, 10, 10, -1]$

Example Explanation1: For 4, the next greater element towards its right is 5.

For 5 and 2, the next greater element towards their right is 10.

For 10, there is no next greater element towards its right.

Example Input2: $A = [3, 2, 1]$

Example Output2: $[-1, -1, -1]$

Example Explanation2: As the array is in descending order, there is no next greater element for all the elements.

Actual Code

```
public class Solution {
```

```

public ArrayList<Integer> nextGreater(ArrayList<Integer> A) {
    Stack <Integer> st = new Stack <>();
    ArrayList<Integer> res = new ArrayList<>(Collections.nCopies(A.size(), -1));
    for(int i =A.size()-1; i>=0; i--){
        while(!st.isEmpty() && A.get(st.peek()) <= A.get(i)){
            st.pop();
        }

        if (!st.isEmpty()) {
            res.set(i, A.get(st.peek()));
        }
        st.push(i);
    }

    return res;
}

```

6.Problem Statement- Sort stack using another stack

Given a stack of integers **A**, sort it using another stack. Return the array of integers after sorting the stack using another stack.

Problem Constraints $1 \leq |A| \leq 5000$

$0 \leq A[i] \leq 10^9$

Input Format: The only argument is a stack given as an integer array A.

Output Format: Return the array of integers after sorting the stack using another stack.

Example Input 1: A = [5, 4, 3, 2, 1]

Example Input 2: A = [5, 17, 100, 11]

Example Output 1: [1, 2, 3, 4, 5]

Example Output 2: [5, 11, 17, 100]

Actual Code

```

public class Solution {
    public ArrayList<Integer> solve(ArrayList<Integer> A) {
        Stack<Integer> tempStack = new Stack<>();
        Stack<Integer> inputStack = new Stack<>();

        // Push all elements of A into inputStack
        for (int num : A) {
            inputStack.push(num);
        }
    }
}

```

```
while (!inputStack.isEmpty()) {  
    int current = inputStack.pop();  
  
    // Move elements from tempStack back to inputStack if they are greater than current  
    while (!tempStack.isEmpty() && tempStack.peek() > current) {  
        inputStack.push(tempStack.pop());  
    }  
}
```

```
    // Place the current element in the correct position in tempStack  
    tempStack.push(current);  
}  
  
// Transfer sorted elements back to an ArrayList  
ArrayList<Integer> result = new ArrayList<>();  
while (!tempStack.isEmpty()) {  
    result.add(tempStack.pop());  
}  
  
// Reverse the result to maintain sorted order from smallest to largest  
Collections.reverse(result);  
return result;  
}  
}
```