STACKS AND QUEUES

STACK

```
class Stack{
    int size = 1000
        top = -1

void push(int n){
        ~~top++;~~
        if (top == size-1) return;
        top++;
        arr[top] = n;
    }

void pop(){
    int n = arr[top];
    top--;
    return n;
  }
```

QUEUE
↓

```
class Queue {
    arr[ ]
    start = -1
    end = -1
    currSize = 0;

  push ( element ) {
     if ( currSize == maxSize ) return

     if ( end = = -1 ) {
            start = 0
            end = 0
      } else {
           end = (end + 1) % maxSize;

     }

     arr[end] = element
     currSize++;
   }
```

← PUSH

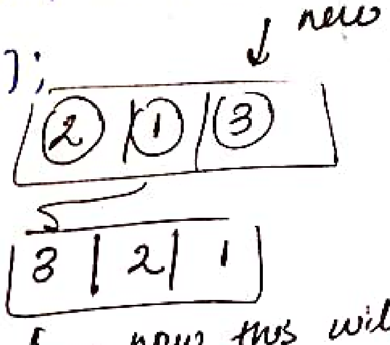
STACK USING QUEUE

Approach

→ push(n) → push the element in the queue

use a for loop of size()-1, remove element from queue and again push back to the queue. hence the most recent becomes the most former element.

```
class Stack{
 Queue<Integer> q = new LinkedList<>();

 void push(int n){
      q.add(n);
      for (int i=0 ; i< q.size()-1 ; i++){
           q.add(q.remove());
      }
 }

 int pop(){
      return q.remov();
 }

 int top{
    return q.peek()
 }
```



now this will get removed first recent one LIFO

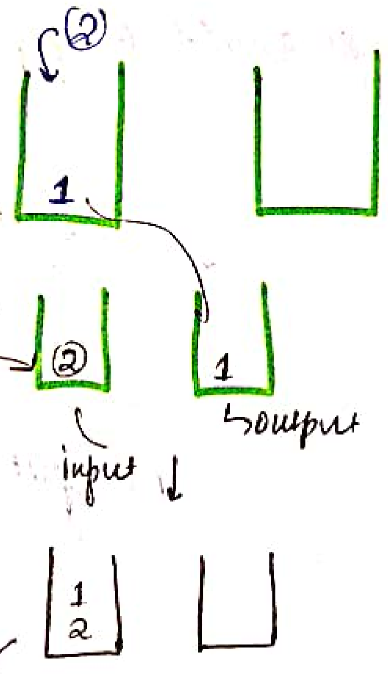
← returns null when empty...

```
public int pop() {
    if (start == -1) return
    int popped = arr [start];
    if (currSize == 1) {
        start = -1;
        end = -1;

    } else {
        start = (start + 1) % maxsize;

    }
    currsize--;
    return popped;

}

public int top() {
    if (start == -1) { return }
    return arr[start];

  }

}
```

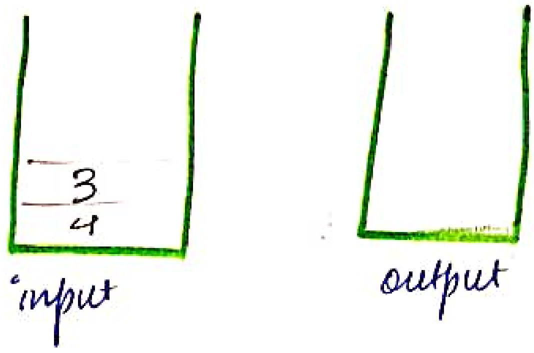
...QUEUE USING STACK...

Approach 1

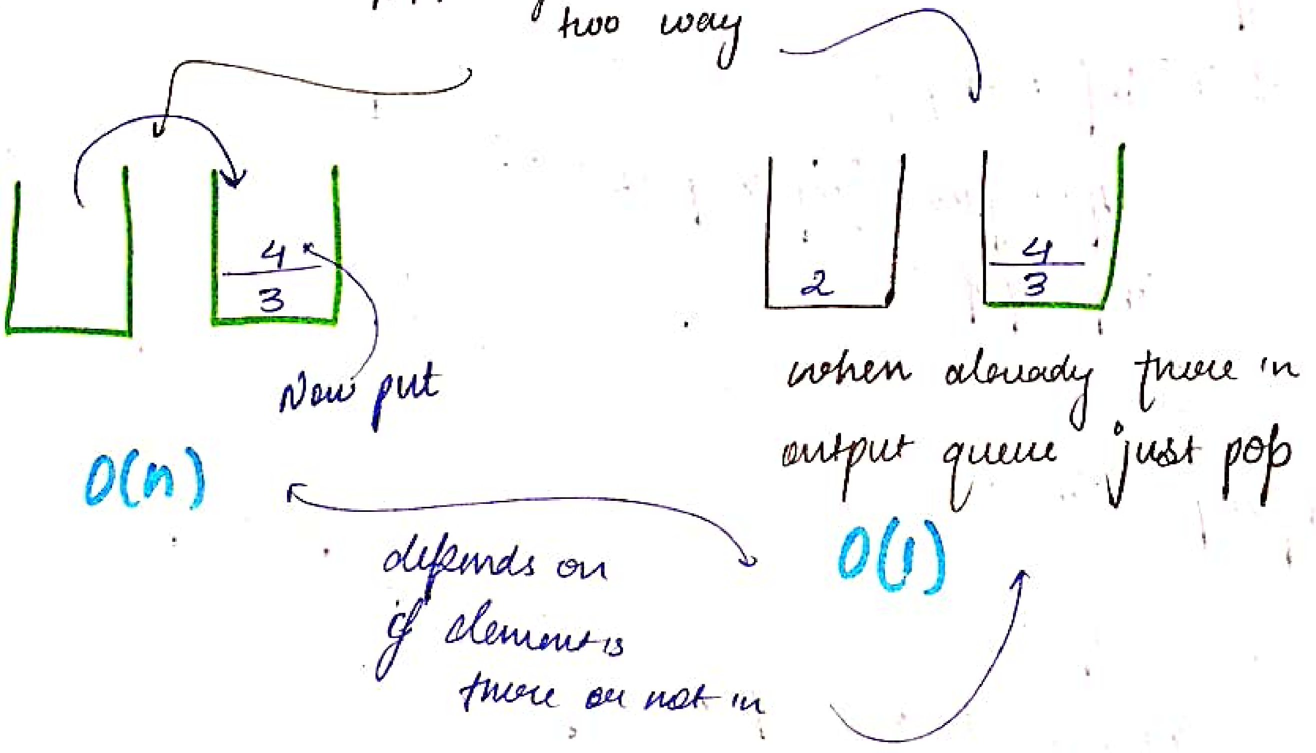
```
clss Queue{
  Stack <Integer> input = new Stack<>();
  Stack <Integer> output = new Stack<>();

                    ✓ O(n)

  void push(n) {
    while( input.empty() == false) {
        output.push (input.peek());
        input.pop().
    }

   input.push(n);

   while (output.empty() == false) {
        input.push (output.peek());
        output.pop ();
   }
  }
 }

 int
 void  pop() {
     if (input.empty()) { return }
        int val = input.peek()
          input.pop();
          return val;
 }
```



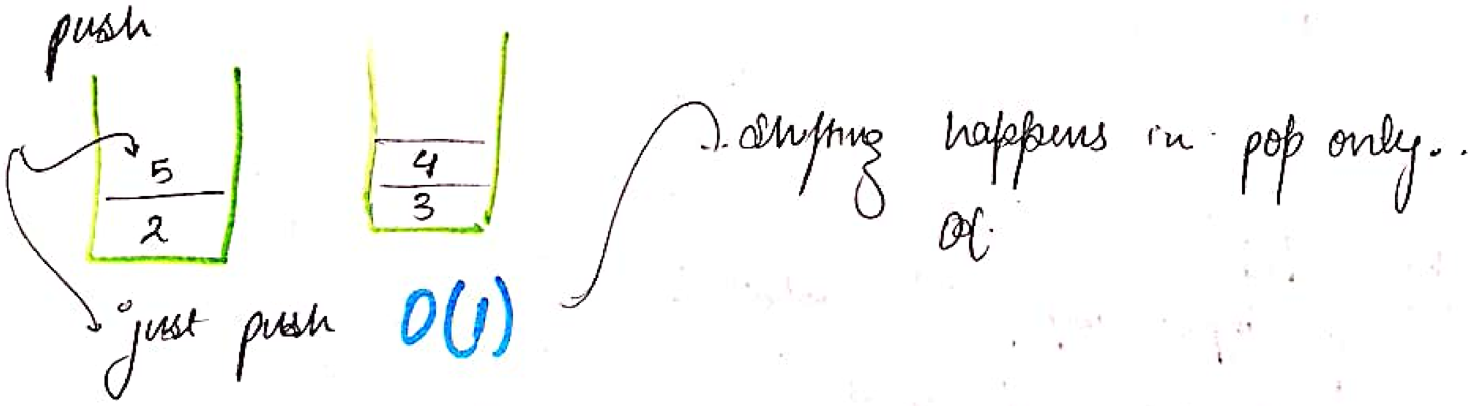
APPROACH : 2



pop : 4 should be popped first ∴ two way

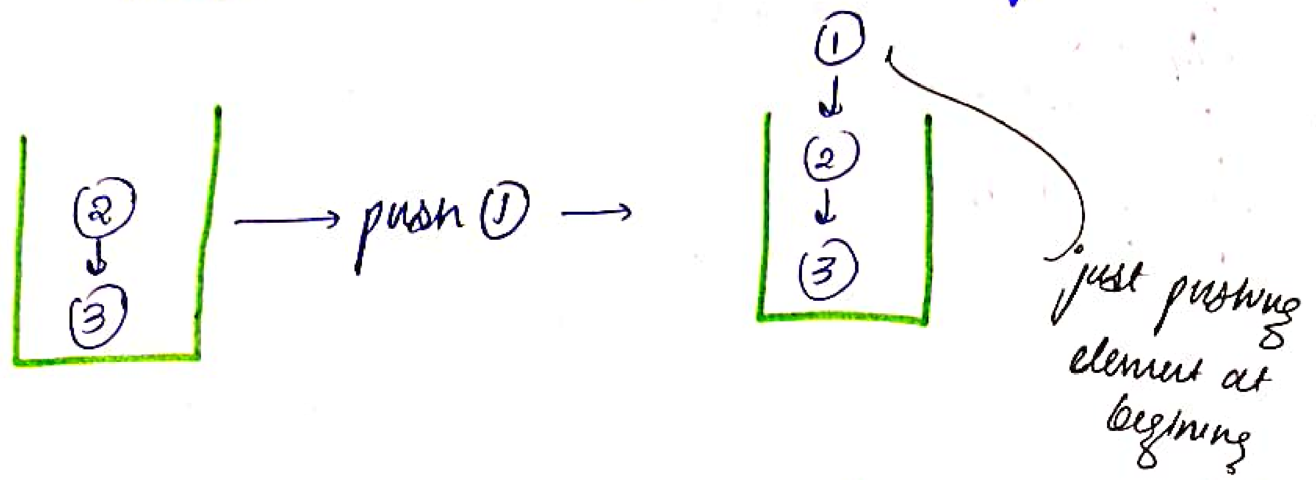

4
3
Now put
O(n)
2
4
3
when already there in output queue just pop
O(1)
depends on if element is there or not in

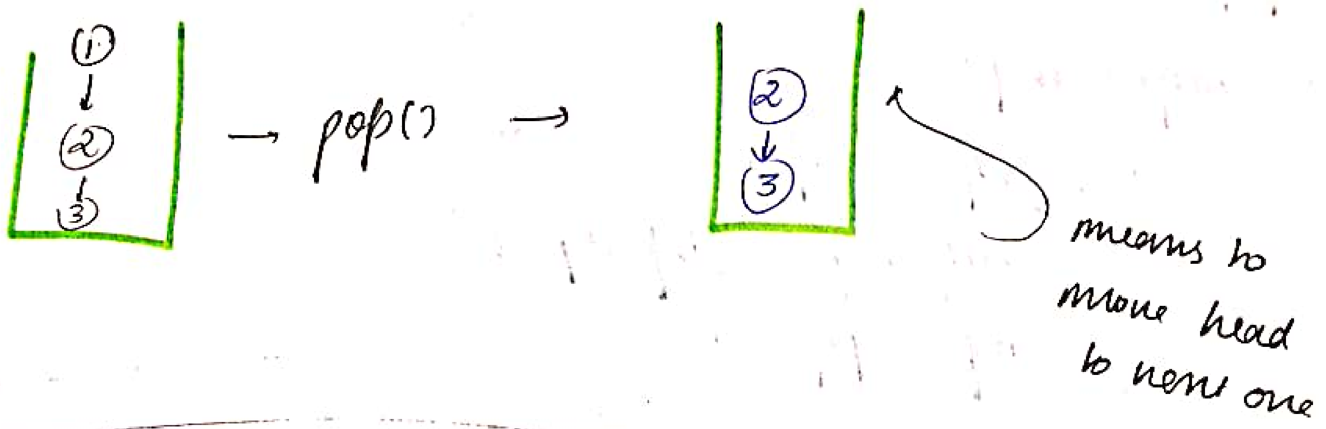
push



```
MyQueue {
      input }  ⟶  2 stacks
      output
  void push (int n) {
     input.push (n);
   }

int pop() {                          ∫ O(n) or O(1)
   if ( output. empty()) {
        while (input.empty == false) {
              output.push( input.peek ))
              input.pop() ;
        }

   int n =  output.peek()
   output.pop()
   return n;                int peek() {
                                if (output.empty()) {
 }                                   shift input element to
                                        output  ←
                                 }
                                return  output.peek()   O(n) or
                                                           O(1)
                             }
  int size() {
        return  output.size() + input.size();
    }
```

...STACK USING LINKED LIST...

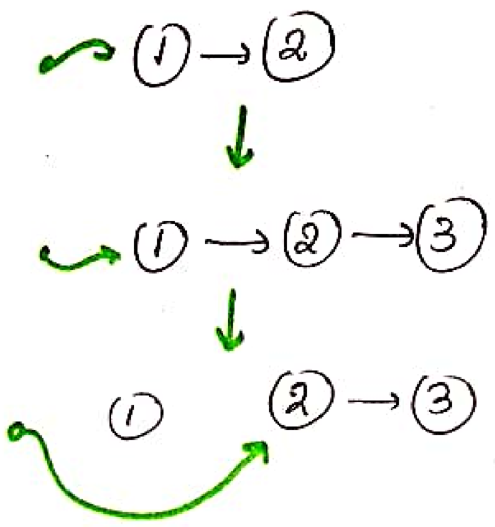




```
Stackpush (n) {
  StackNode element = new StackNode(n);
  element.next = top;
  top = element;
  size++;
}
```

```
StackPop(){
  if(top==null) return -1;
  int topData = top.data;
  StackNode temp = top;
  top = top.next;
  return topData;
  }
```

QUEUE USING LINKED LIST



```
class Queue {
    front = rear = null;
    size = 0

 void enqueue(value){
    QueueNode temp;
    temp = new QueueNode(value);
    if(temp == null) {
                          full
                     }

    else {
        if ( front = null) {
                   front = rear = temp;
               }

        else {
           rear.next = temp;
           rear = temp
         }
 }                        size++   }
```

```
void dequeue(){
  if(front==null){
             empty
        }

  else{
      QueueNode temp = front;
        front = front.next;
        size--;

      }
}
```

INFIX TO POSTFIX

✓ Algorithm steps

→ initialize an empty stack (for operators)

→ initialize an empty result (postfix expression)

→ scan expression from left to right

→ for each token:

- operand → add directly to postfix expression
- left '(' → push onto stack
- right ')' → pop until '(' is encountered and add to postfix and discard both parentheses.

- operator (+, -, *, /, ^):

  if precedence of current operator $\leq$ precedence of top of stack pop and top should not be '('

  push operator onto stack

• after expression pop all remaining operator from stack to ~~prefix~~ postfix

			Associativity
$\wedge$	$\rightarrow$ 3	$\rightarrow$	right to left
$*$ / %	$\rightarrow$ 2	$\rightarrow$	left to right
+ -	1	$\rightarrow$	left to right.

Associativity

$\underbrace{A-B}-C$

↓ first this
then
$\underbrace{Z-C}$
this (left to right evaluation when precedence equal)

~~A~~ $2^\wedge\underbrace{3^\wedge4}$ — $234^{\wedge\wedge}$ ~~or~~ not ~~$2^\wedge3^\wedge4$~~ $23^\wedge4^\wedge$ X

↓ first this

$\underbrace{2^\wedge81}$
then this (right to left evaluation when precedence is equal)

* otherwise wrong output

INFIX TO PREFIX

$(A+B) * C - D + F$

# Reverse the given infix

# Do infix to postfix conversion
- do not pop when precedence is equal
- we only pop when right associative operator

# Reverse the answer

$(A+B)* C - D + F$

$F + D - C *) B + A C$ opening $\longleftrightarrow$ closing

$F + D - C * (B + A)$

↓

postfix

$\rightarrow$ FDCBA+*-+

↓

reverse

+-*+ABCDF

★ Associativity check

```
while (!stack.isempty && precedence(curr) <=
        precedence(stack.peek()) &&
        isLeftAssociative(curr)) {
            postfix.append(stack.pop());
}
```

element pop only when left to right associativity

```
isLeftAssociative(char ch) {
        return ch != '^';
}
```

O(n) — time

O(n) — SPACE

★

```
while ( ! stack.isEmpty() &&
        precendence(stack.peek()) > precendence (curr) ||
(precendence(stack.peek()) == precendence (curr) &&
    isRightAssociative (currentop) ) {
        result.append( stack.pop );

}
```

when == then associativity should be right

why?

we scan reverse the expression

∴ associativity gets flipped

left (+, -, *, /) becomes right

right (^) become left

$2^3^4 \rightarrow$ we do not pop as exp evaluates to right to left

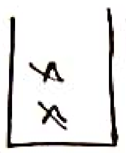
↓

$4^3^2 \rightarrow$ we need to pop as ~~right at~~ associativity changes and needs to evaluate 4^3 first.

$2^\wedge3^\wedge4$

$\downarrow$

$4^\wedge3^\wedge2 \rightarrow \boxed{43^\wedge2^\wedge} \rightarrow {}^\wedge2^\wedge34$ ✓



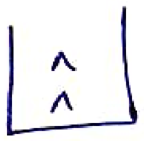
but when not poped

$2^\wedge3^\wedge4$

$\downarrow$

$4^\wedge3^\wedge2 \rightarrow \boxed{432^\wedge{}^\wedge} \rightarrow {}^\wedge{}^\wedge\underbrace{23}_{\text{during evaluation}}4$ ✗

this will be evaluated first wrong



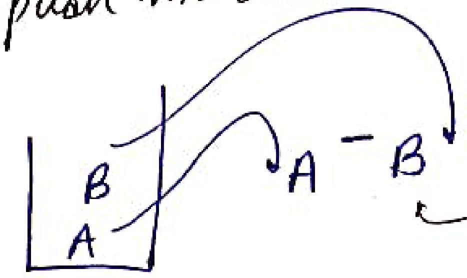
$\text{time} - \underbrace{O\left(\frac{n}{2}\right) + O\left(\frac{n}{2}\right)}_{\text{reverse}} + O(2N)$

$\text{SPACE} - O(N)$

POSTFIX TO INFIX

AB - DE + F * /

scan left to right
- operand are pushed onto stack
- when operator cor pop two operand combine and push into stack



first popped should be on right

A B - D E + F * /

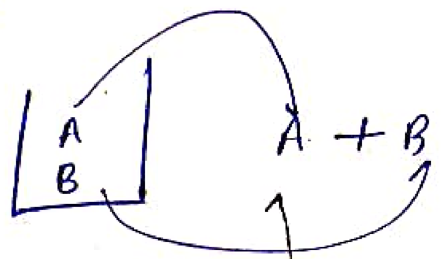
	St
A	A
B	A B
-	(A - B)
D	(A - B), D
E	(A - B), D, E
+	(A - B), (D + E)
F	(A - B), (D + E), F
*	(A - B), ((D + E) * F)
/	((A - B) / ((D + E) * F))

empty stack

PREFIX TO INFIX

Scan from right to left
Same as before

but A



first popped will come in front.

POSTFIX TO PREFIX

AB - DE + F * /

pattern = (operator) (top2) (top1)

	st
A	A
B	AB
-	-AB
D	-AB, D
E	-AB, D, E
+	-AB, +DE
F	-AB, +DE, F
*	-AB, ~~+~~ *DEF
/	/-AB*DEF

PREFIX TO POSTFIX

/-AB*+DEF

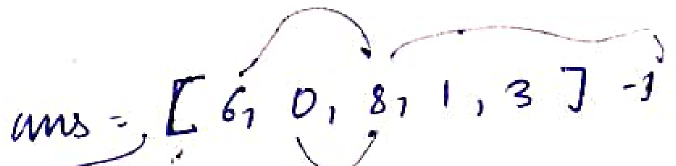
scan from right to left

pattern = (top 1) (top 2) (operator) ★

	St
~~A~~	~~A~~
F	F
E	F, E
D	F, E, D
+	F, DE+
*	DE+F*
B	DE+F* , B
A	DE+F* , B, A
-	DE+F*, AB-
/	AB- DE+F* /

NEXT GREATER ELEMENT

MONOTONIC STACK → when elements are stored in specific order.



ans = [6, 0, 8, 1, 3] -1

tell the next greater

[8, 8, -1, 3, -1]

BRUTE

# iterate and find next greater $\int O(n^2)$

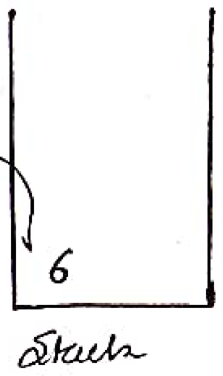
```
for (i=0 → n-1) {
        for ( j=i+1) → {
                if (arr[j] > arr[i])
```

OPTIMAL

# traverse from back

4	12	5	3	1	2	5	3	1	2	4	6

-1

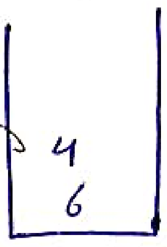


|4|6|

↑ in stack theres 6

∴ next greater = 6

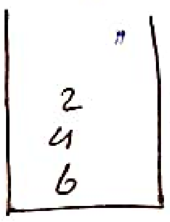
push 4 in stack



|2|

↳ in stack (4) nge

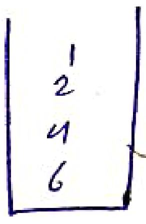
and push 2 in stack

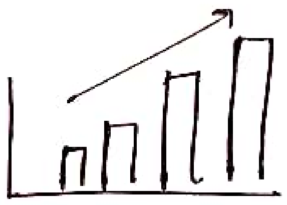


|1|

stack (2) nge

and push ~~(2)~~ (1)

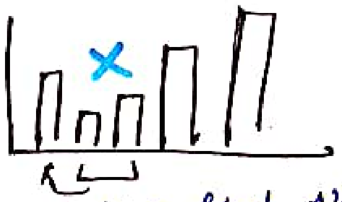




|3|

✱ time find next greater

if we insert 3 directly then



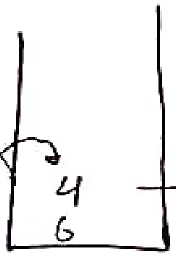
3 ✱ to find NGE

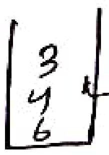
2 and 1 do not matter ∴ pop and maintain order

3

3 nge → 4

and push 3





we do not need 1 and 2

main specific order
we do not need middle

smaller one

to find nge

$\int O(n)$

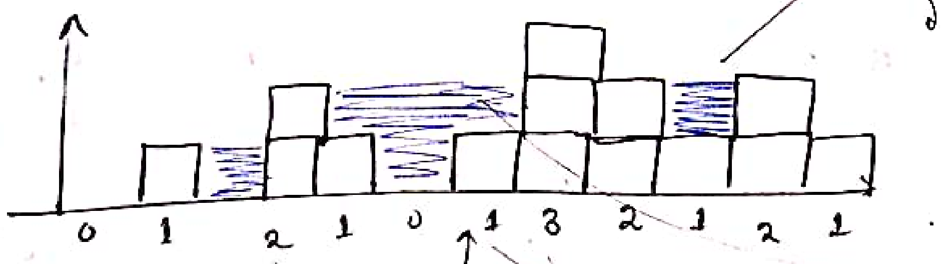
```
findNGE ( ) {
    for (i = n-1) -> 0 ) {
        while ( !st.empty() && st.top <= arr[i] ) st.pop();
        if (st.empty()) nge(i) = -1
        else   nge[i] = st.top()
        st.push(arr)
    }
    return nge
}
```

Time - O(2N)

when last element
is biggest one
and at more
only n element
can be removed
overall
∴ N+n = 2N

SPACE → O(n)

TRAPPING RAIN WATER



6 units of water

BRUTE

for every index

$\sum$ min (leftMax of RightMax) − arr[i] = 1 unit

< arr[i]   < arr[i]

```
total = 0
for (i = 0 ——→ n-1) {
    if (arr(i) < leftMax && arr(i) < rightMax) {
        total += min( , ) - arr(i)
    }
    return total
}
```

leftMax = prefixMax [i] [i]

rightMax = suffixMax [i]

prefixMax[n]
↑
max till that index

arr = [2, 1, 0, 5, 3]

prefix = [2, 2

[2, 2, 2,

[2, 2, 2, 5,

→ [2, 2, 2, 5, 5]

we have got max to left of index

suffix[n]

arr = [2, 1, 0, 5, 3]

suffix = [3]

[0, 5, 3]

[5, 5, 3]

[5, 5, 5, 3]

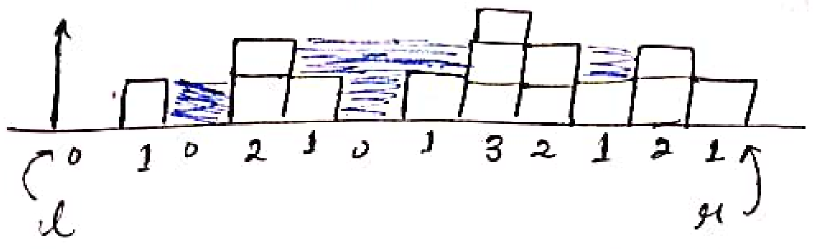
[5, 5, 5, 5, 3]

```
prefix[0] = arr[0]
for (i=1 → n-1){
    prefix[i] = max (prefix[i-1], arr[i])
}
```

time → O(3N) → O(N)

SPACE → O(2N)

OPTIMAL

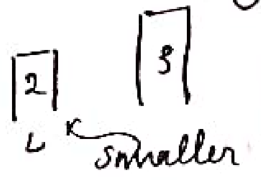


now we only need length of smaller one.

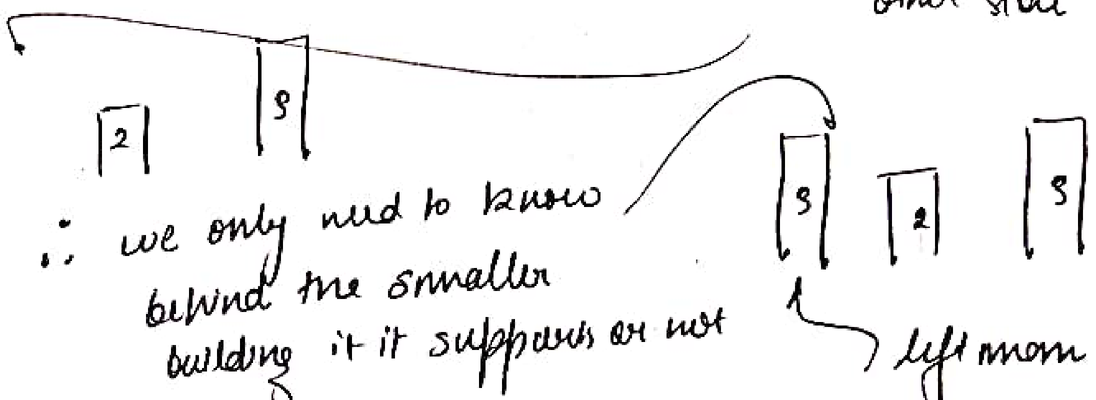
lmax = 0 ← left main biggest building

rmax = 0 → right main biggest building

now during traversal one of them would be smaller so ~~wonto~~ know that there is surely a taller building on the other side

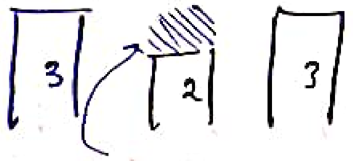


∴ we know we have the support from other side



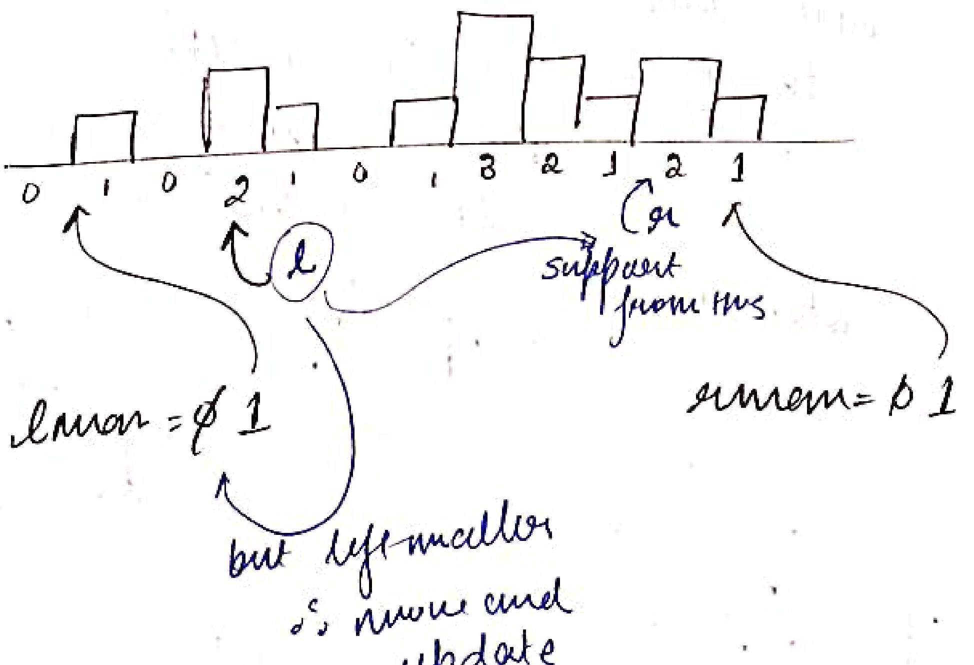
∴ we only need to know behind the smaller building it it supports or not

∵ leftmax ≥ smaller one
∴ it can store water



concept

traverse two building figure out smaller one we know opposite side supports us ∵ check behind if the it supports or not ∴ calculate water unit is supporter
after this update right or left max according to L or R building



..SUM OF SUBARRAY MINIMUM..

arr = {3, 1, 2, 4}

# sum all the min elements in all subarray

{3} {1} {2} {4} {3, 1} {3, 1, 2} {3, 1, 2, 4}

{1, 2} {1, 2, 4} {2, 4}

# 3 + 1 + 2 + 4 + 1 + 1 + 1 + 1 + 1 + 2 = $\boxed{17}$

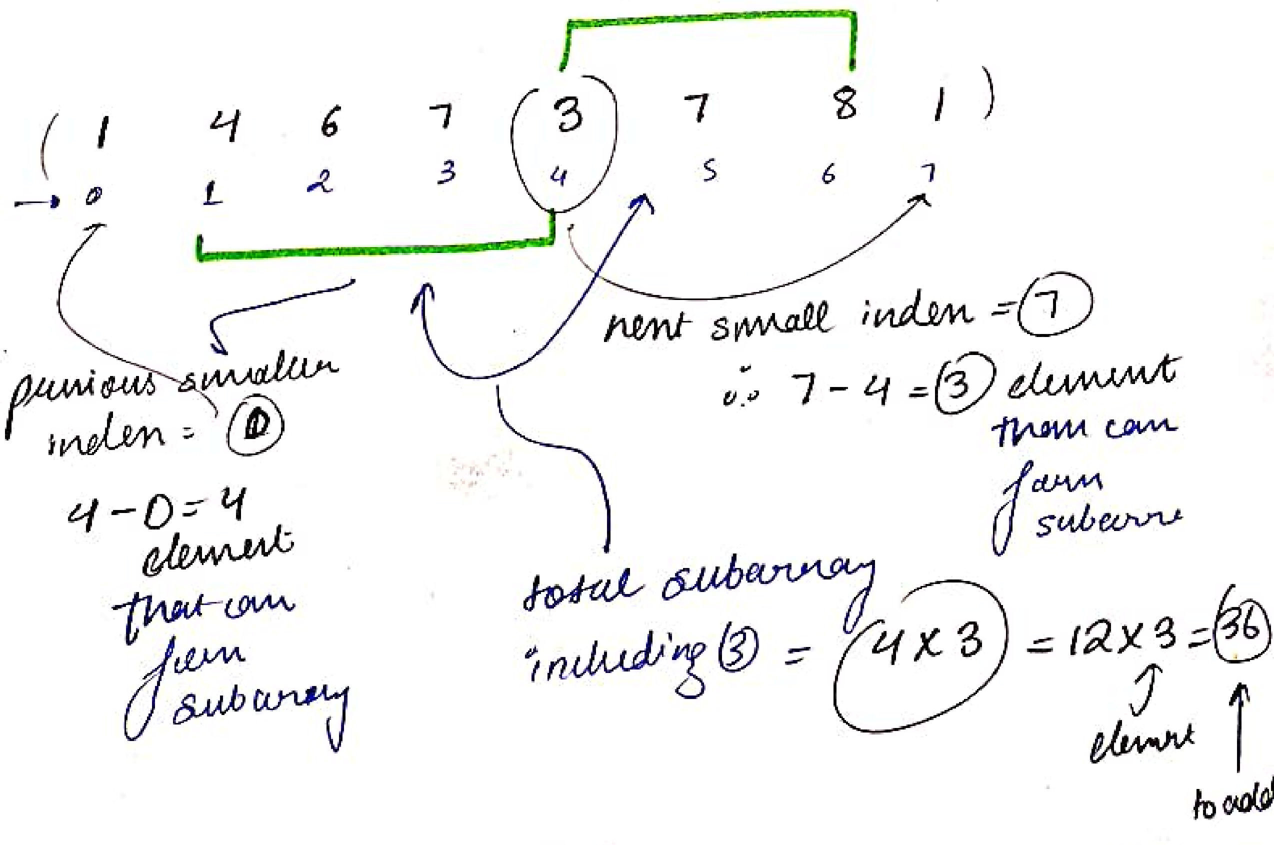
BRUTE

→ Generate all subarray

→ figure out minimum while extending array

↗ $O(n^2)$

OPTIMAL



(1 4 6 7 3 7 8 1)
0 1 2 3 4 5 6 7
previous smaller index = 0
4 − 0 = 4 element that can form subarray
next small index = 7
∴ 7 − 4 = 3 element them can form subarray
total subarray including 3 = 4 x 3 = 12 x 3 = 36
element
to add

```
int  findtotal (arr)
   lmax = 0 , rmax = 0,  total = 0, l = 0, r = n-1
   while (l<r){            ∫ support from right
      if[arr[l] <= arr[r]){
          if( lmax > arr[l]) {
              total = lmax - arr[l]
          }
          else lmax = arr[l]
          l = l+1
      }
   else{
       if (rmax > arr[r]) {
           total = rmax - arr[r]
       }
       else rmax = arr[r]
           r = r-1
      }        ⤳ O(n)
  }
  return total
  }
```

Edge case when element equal

arr = [1 , 1]
       0   1

nse → 2   2

pse → -1   -1

0 index = left = $0-(-1) = 1$
right = $2-0 = 2$] $1\times 2 = \boxed{2}$

[1 .]
[1, 1]

but

1 index left = $1-(-1) = 2$
right = $2-1 = 1$] 2 = [1, 1] *

this included twice

∴ only consider
left one or right one
∴ do not consider on both

when pse → do not look < but
include = (equal) also
∴ psmaller or equal

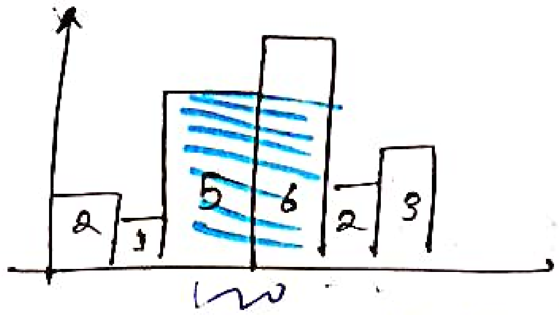
1   1
0   1

nse → 2   2
pse → -1 $\boxed{0}$ ← as it considers this also

∴ # removes duplicate when equal

```
int sum(arr){
      ~~nes~~
      nse = findNse(arr)
      psee = findPSEE(arr)
      total=0 , mod = (int)(1e9+7)

  for (i=0 → n-1){
        left = i - psee[i]
        right = nse[i]-i;
        total = (total + (right × left × arr[i])%mod)%mod

    }
  return total
```

... LARGEST RECTANGLE IN HISTOGRAM ~



5+5 = 10 being the largest

BRUTE

go to every index

2, 1, 5, 6, 2, 3
↓
it cannot go left or right ∴ 2×1 = 2

2, 1, 5, 6, 2, 3
↑
cannot go to all ∴ 1×6 = 6

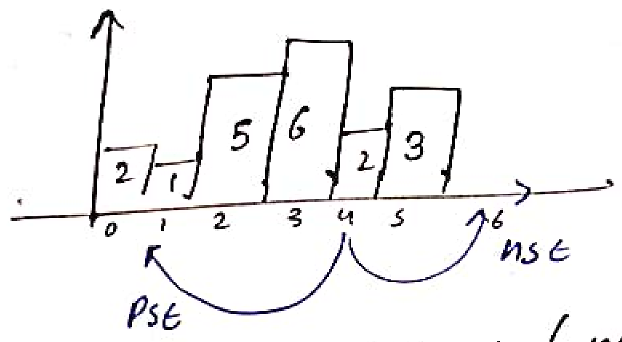
2, 1, 5, 6, 2, 3 ∴ 5×2 = 10
↑
cannot go to
6 only ∴

⋮

similarly other

∴ we can go the smaller element only

∴ we need (pse, nse)

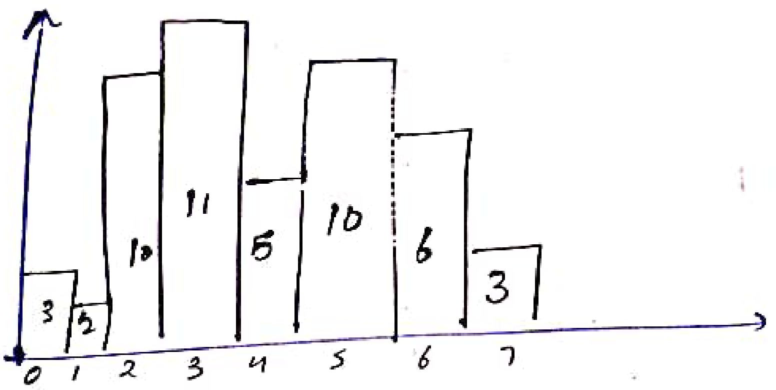


arr[i] × (nse − pse − 1)
↖ height   ↖ width

pse, nse
↙ −1   ↘ n

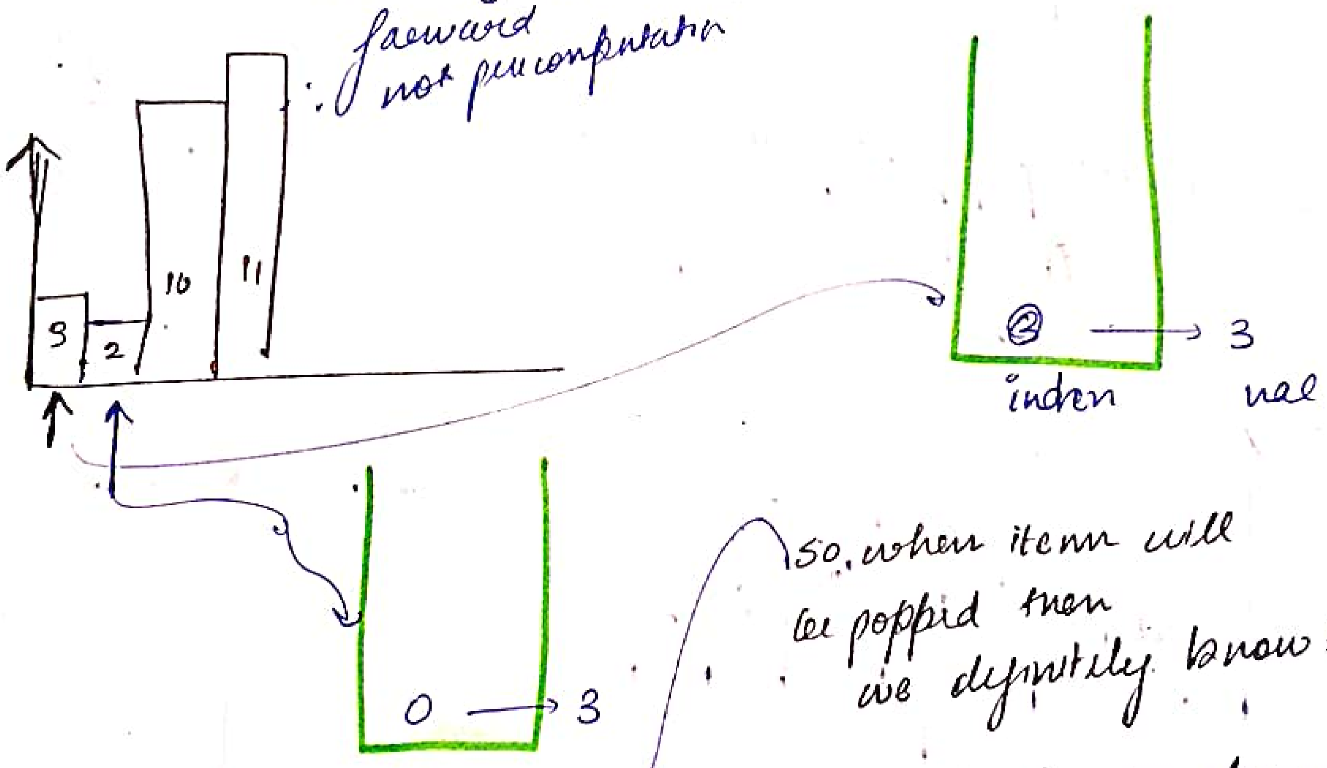
```
fun (arr)
    nse = findNSE (arr)   ⟶   2×n
    pse = findPSE (arr)
    max = 0
    for (0 ⟶ n−1) {                    ↙ O(n)
        area = arr[i] × (nse(i) − pse(i) − 1)
        max = Math.max (area, max);
    }
    return max;
}
```

time − O(n)
space → O(n)



arr[i] × (nse − pse − 1)

this can be computed while going forward ∴ not precomputation



will pop ③ as it is smaller

so, when item will be popped then we definitely know that popped nse element is current one "2" as it is smaller the top

popped pse is popped element below in the stack
if empty then '-1'

current's index

∴ 3 nse → ~~2~~ 1

popped

3 pse → -1 as stack empty

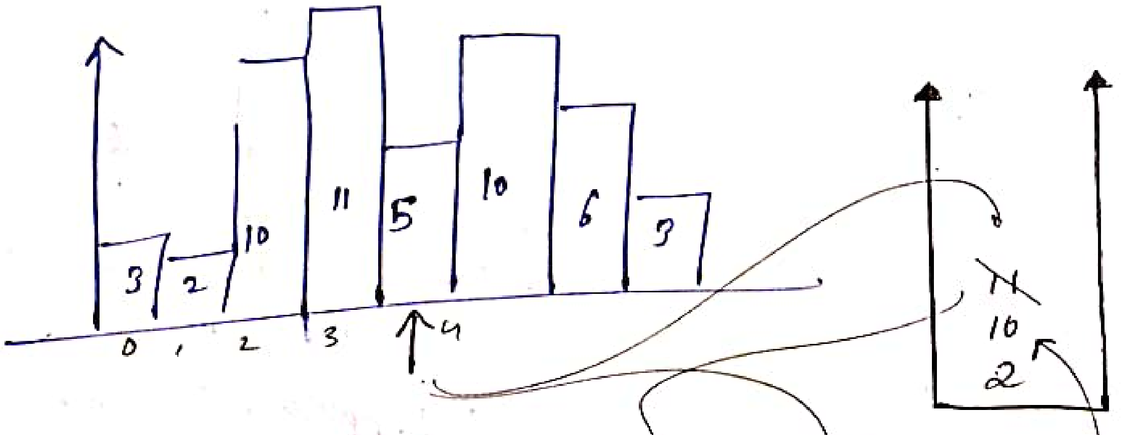
put in formulae

arr(i) x (nse - pse - 1) = area

3 x (1 - (-1) - 1) = area

3 x (1) = $\boxed{3}$

∴ while popping / coming back we know that the nse and pse ∴ compute only then



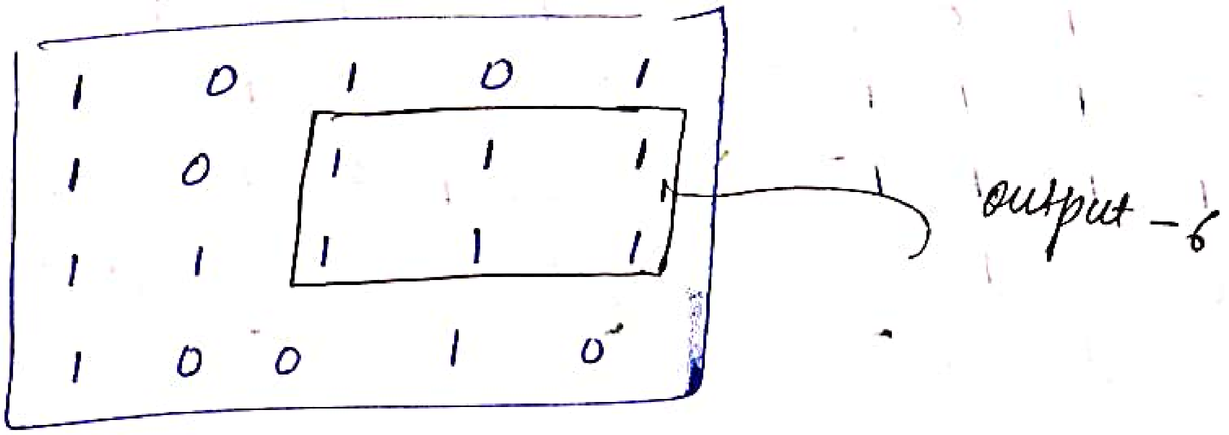
11 nse 5 → 4

11 pse → 10

when element still in stack
then nse is "N" as no next smaller one
and all PSE is stack empty the "-1"

```
fun(arr){
    Stack st, maxArea = 0        O(n)
    for(i=0 ⟶ n-1){
                                 index
        while(!st.empty() && arr[st.top] > arr[i]){
            element = st.top()
            st.pop()
            nse = i
            pse = st.empty ? -1 : st.top
            area = element x (nse - pse - 1)
            max = max(area, max)        popping throughout
        }                               O(n)
        st.push(i)
    }
    while(!st.empty()){
        nse = n
        pse = st.empty() ? -1 : st.top
        area                     time - O(2N)
        and update               space - O(N)
    }
    return maxArea
}
```

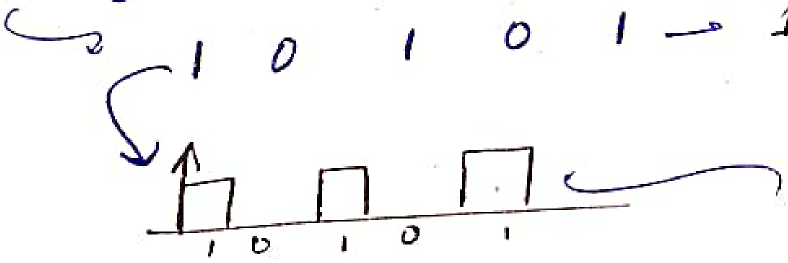
MAXIMAL RECTANGLE



give area of max rectangle filled with 1s

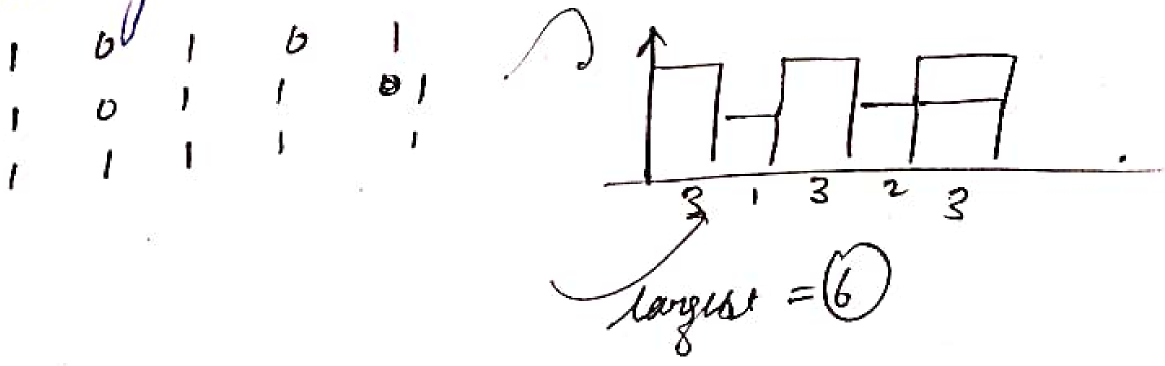
using histogram approach

1 0 1 0 1 → 1st row



find max rectangle here

similarly do with other rows



precompute height

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 0 & 1 & 1 & 1 \\ 2 & 0 & 2 & 1 & 2 \\ 3 & 1 & 3 & 2 & 3 \\ 4 & 0 & 0 & 3 & 0 \end{bmatrix}$$

$\uparrow$ $O(n^2)$

fun X(mat()()) {

  n, m, psum()()

  ~~sa~~ sum=0

  fun (j=0 ⟶ m-1) {

histergam x O(mn)

time - $O(m \times n) + O(n \times 2m)$

space - $O(n^2)$