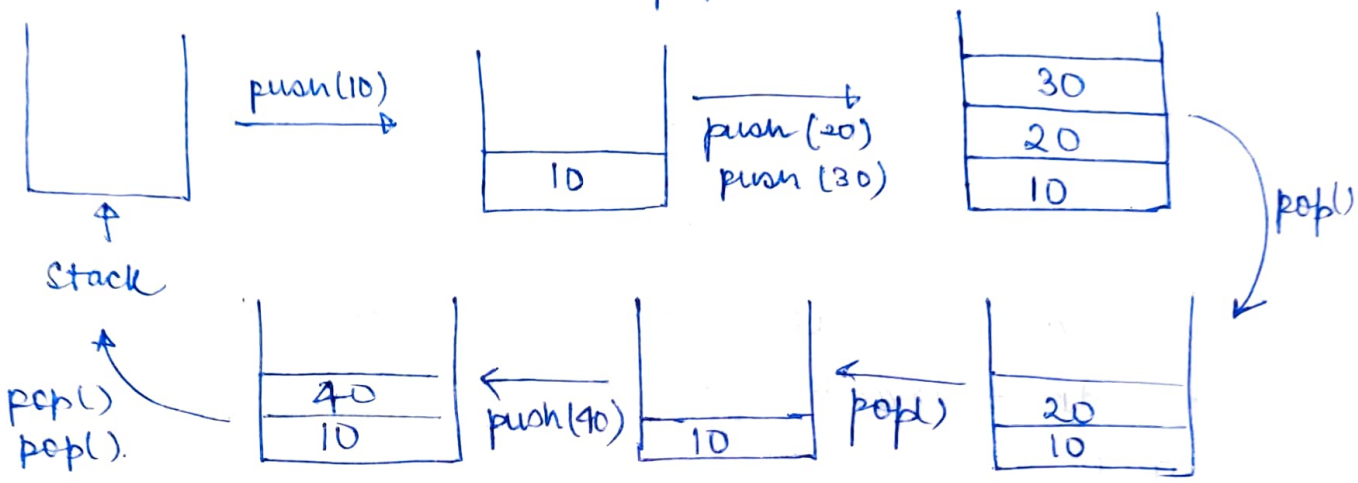
Stacks (Last In first out)

- can be imagined as a str box closed at one end
- Insertion opo operation is called push operation and removal is called pop



Stack Operations

- isEmpty() : Returns true if stack is empty
- push(x) : Inserts an item x to the stack.
- pop() : Removes an item from the top of stack.
- peek() : Returns the top item. (similar can be achieved by top();)
- size() : Returns the size of stack.

* ~~heo~~ Linear Data Structure
* Abstract Data type → The behaviour of operations is fixed. For ex- In stacks element must be pushed and poped in the end only.

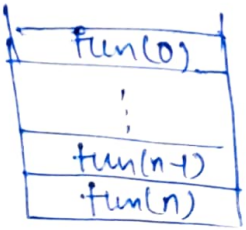
Ex:- Recursion uses Stack

fun(n)

for calculating factorial (n)

fun(n+1)

fun(n-2)



Corner conditions

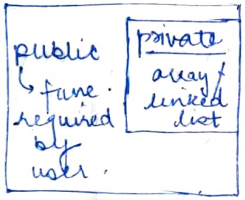
(a) **underflow** : when stack is empty and user calls pop() or peek / top functions.

(b) **overflow** : when push is called on a full stack.

Array Implementation using Array

Functions required → (a) push (x)
(b) pop()
(c) top() / peek()
(d) size()
(e) isEmpty().

- We use class to implement this functionality
- As we don't want user to ~~know~~ access the full data structure
- we make the array private and make some operations on it, so that ~~array~~ user can implement some functionality



- If we're working with array, we have to mention ~~the~~ its size.
- So in this implementation user has to specify the size of stack.
- How to maintain the pos of last item:
  We'll create a variable which always points to the last element of stack.
- We'll dynamically create our Array.

Ex :- int * arr;

private prop.

After user has defined the size we just create an array of defined size dynamically

arr = new int [size]

First we have to define constructor

```
MyStack (int capacity) {
    cap = capacity;
    arr = new int [cap];
    top = -1;
}
```

cap → private member
top
arr
For defining the size of stack
To keep track to last element

Functions

(a) **pushing an element**

```
void push(int x){
    top ++;
    arr [top] = x;
}
```

if (top == cap-1) { cout << "OF"; return; }

(b) **poping an element**

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

if (top == -1) { return 0; }

(c) **peeking an element**

```
int peek (){
    return arr[top];
}
```

if (top == -1) { cout << "Array is empty"; return; }

(d) **size of stack**

```
int size (){
    return (top+1);
}
```

(e) **bool isEmpty(){**

```
return (top == -1)
}
```

* The function does not include overflow and underflow conditions
  ⊙ Red shows underflow and overflow conditions.

Time complexity → O(1) in insertion deletion & peeking

Array Implementation

```
Class MyStack {
private: int * arr;
         int cap;
         int top;
         MyStack(int c) {
             cap = c;
             top = -1;
         }
             arr = new int [c];
         }
public:  void push (int x) {
             if (top == cap-1) {
                 cout << " Stack is
                         full";
                 return;
             }
             top++;
             arr[top] = x;
         }
         void pop() {
             if (top==-1) {
                 cout << " Stack is empty";
                 return ;
             }
             int res = arr[top];
             top--;
             return res;
         }
         ;
         ;
```

simply one
renaming func.

Vector Implementation

```
class MyStack {
    private:
        vector <int> v;
    public :
        void push(int x){
            v.push_back(x);
        }
        int pop() {
            int res = v.back();
            v.pop_back();
            return res;
        }.
        int size() {
            return v.size();
        }
        int peek() {
            return v.back();
        }
        bool isEmpty(){
            return v.empty();
        }
```

Linked List Implementation of Stack

- In case of arrays all the operations i.e push, pop, peek ... were performed in O(1) complexity except for 1 operation in case of dynamic arrays which includes copying of data from old to new array.

Implementation of Dynamic Sized stack using array

* The only change will be
  1. we don't need to input stack size from the user
  2. we have to modify the push function as there will be no overflow condition

constructor modified

```
stacks() {
    cap = 4;
    ;
}
```

defined by us

push function

```
void push(int data) {
    if (top == cap-1) {
        int * newData = new int[cap*2];
        for (int i=0; i<cap; i++) {
            newData[i] = arr[i];
        }
        delete [] arr;
        cap *= 2;
        arr = newData;
    }
    top++;
    arr[top] = data;
}
```

copying element to the another array of size doubled

Implementation using linked list

(1) using standard technique of insertion and deletion in linked list

↓

peek() + isEmpty() + size() — O(1) { we'll maintain a tail pointer so that we can perform insertion operation in O(1) time

↓

pop() function

maintaining a previous pointer

25 → 35 → 45 → null

prev ↑ tail ↑

pop();

25 → 35 → null

tail ↑

prev ↑

previous is at same because we can't traverse backwards in linked list

✗

Wrong approach

traversing from the node just behind tail

25 → 35 → 45 → null

head ↑ tail ↑

using temp.

temp = 35

by the help of temp we can delete 45 & update tail also

This states that pop function will be implemented in O(n) time

which is very costly for stack.

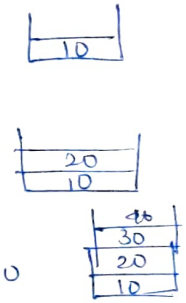
Effective Approach

* Insert element at beginning

We know that insertion and deletion at head of linked list is O(1). So we can use this concept here.

Ex:

- push(10) — 10 ← head
- push(20) — 20 → 10 ← head
- push(30) — 30 → 20 → 10 ← head
- push(40) — 40 → 30 → 20 → 10 ← head
- pop() — 30 → 20 → 10 ← head



* Every operation is of O(1)

⊙ For the size() function we have two approaches

- traversing till null & get size — O(n)
- maintaining a private data variable size and increase and decrease at push and pop respectively → and just return size when user calls size function

```
void push(int x) {
    Node * newNode = new Node(x);
    newNode -> next = head;
    head = newNode;
    size++;
}
```

pop(), top(), isEmpty(), size() all can be implemented in O(1).

Templates

- Templates can be used when we want to use the same piece of function / class for different data types.

```
template <typename T>
class Pair {
    T x;
    T y;
  public:
    void setX(T x) {
        this->x = x;
    }
};
```

To avoid compilation errors.

- whenever we've created a class using templates, we have to specify the 'T' while creating objects of that class.

* While creating object we'll specify the datatype

For ex:
```
Pair <int> p1;
Pair <double> p2;
Pair <float> p3;
```

* If we want to use more than 1 datatype in a class, we can assign different variable names to each of them

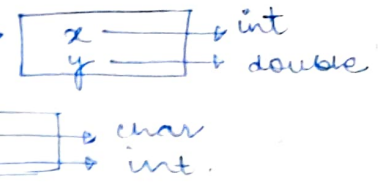
```
template <typename T1, typename T2>
class Pair {
    T1 x;
    T2 y;
  public:
    func;
};
```

while creating objects

```
Pair <int, double> p1
```

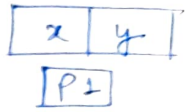
creates a class with
x as int &
y as double.

```
Pair <int, double> p1
Pair <char, int> p2
```

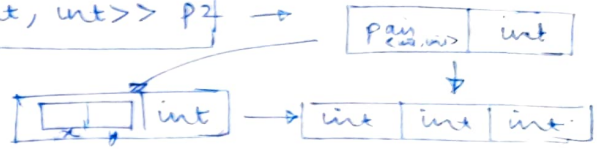


(16) How to create a triplet with a pair class

```
pair<int, int> p1
```




```
pair< pair <int, int>> p2
```




```
pair< pair <int, int >> p2;
p2.getY(); ⟶ gives values of p2.y (simple)
p2.setX(p4)
```

expecting argument of type pair

```
pair<int, int> p4;
p2.setX(p4);
```

creating triplet of
- int
- double
- char

```
pair < int, pair < double, char >> p;
```

STACK IN STL (Container adapter)

- We have to mention the datatype, the stack will be storing

O(1)

```
stack<datatype> s;
```

stack<int> stack<char> stack<float>

- functions

(a) s.push(data) (c) s.pop(); (e) s.empty()
(b) s.top(); (d) s.size();

- The difference is that the inbuilt pop function of stack class is of return type void

```
void pop(){
}
```

Balanced Parenthesis

I/P : {a + (b - c) * d}. → True

I/P : {a + (b - c + [e - f)]} → False

I/P : ((())) → False

I/P : {}[()] → True.

- we can use stack here, just storing the opening brackets in stack and then poping the top element whenever we see a closing one.

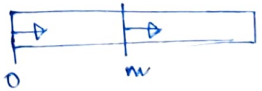
```
bool isBalanced(string str){
    stack<int> s;
    for(int i=0; i<str.length(); i++){
        if(str[i]=='}' || str[i]==')' ||
                str[i]==']'){
            s.push(str[i]);
        }
        else{
            if(s.empty()==true)
                return false;
            else if(matching(s.top(), str[i])==false)
                return false;
            else
                s.pop();
        }
    }
    return (s.empty==true);   // Extra opening brackets
}

bool matching(char a, char b){
    return ((a=='(' && b==')') ||
            (a=='{' && b=='}') ||
            (a=='[' && b==']'));
}
```

Implementing two stacks ~~who~~ with one array

1st Approach

divide the array in two half and then keep 0, m (middle) as starting points of two stacks



- Disadvantages

even if we have 0 element in stack 2 i.e right half is empty. on filling left half we can't push more element in S1

```
class twoStacks{
    int arr[];
    void push1(){--}
    void push2(){--}
    int pop1(){--}
    int pop2(){--}
    int size1(){--}
    int size2(){--}
    int top1(){--}
    int top2(){--}
};
```

All these func are to be implemented

Method-2


```
class TwoStacks {
    int *arr, top1, top2, cap;
    public:
    TwoStacks (int n){                    // constructor
        cap = n;
        top1 = -1;
        top2 = cap;
        arr = new int[cap];
    }
    void push1(int x){                    // pushing element to 1st stack
        if (top1 < top2-1) {
            top1++;
            arr[top1] = x;
        }
    }
    void push2 (int x){                   // pushing element to 2nd stack
        if ( top1 < top2-1){
            top2 ++;
            arr[top2] = x;
        }
    }
    int pop1 (){                          // poping element
        if (top >= 0) {
            int x = arr[top1];
            top1--;
            return x;
        }
    }
}
```

Overflow
top1 ≥ top2

underflow
top1 < 0
top2 ≥ cap

Implement 'k' stacks in an array

- we will use a variable 'sn' to identify the stack we want to operate on
- If there are k stacks then the value of sn will be from $0 \leq sn \leq k-1$

Approach-1

divide array into k equal parts (static partitioning)

⊙ space inefficient

Approach-2

- we'll create one more array as nextIndex -

Initially nextIndex is filled as:-

arr[] → {0, 0, 0, 0, 0, 0}
nextIndex → {1, 2, 3, 4, 5, -1}

nextIndex[i] → it provides the index of next empty slot

Ex:- at ni[1], initially the next available space is '2'

Similarly for ni[5], there is no free space, hence -1

We have another array of size 'k' storing the top index of each stack

topofstack = {-1, -1, -1};

topstack[i] → tells the index of top of i^{th} stack

Next available serves two purpose
① gives the next available index
② provides the previous element index

① initially we have our data array which serves as a container to implement k stacks.

② we have an array called nextIndex of size same as data array which initially points to the next free slot in the data array, hence initially

nextIndex = {1, 2, 3, 4, 5, -1}.

④ NextIndex also serves another purpose. at any index 'i' it points to the previous index filled by that stack

For ex push (2, 7)

suppose nextAvailable = 3.

nextIndex[3] = top(2).

③ nextAvailable :- It points to the next free slot available in the array.

④ top of stack :- It is an array of size 'k', it points to the top of i^{th} stack

topofStack[i] = (m) → index of data[i] where top of i^{th} stack is stored.

freetop = nextAvailable

Ex:- steps while we push anything to stack

① check for nextAvailable slot

② put the data into arr[nextAvailable].

③ now modify freetop to next[freetop]

④ next[i] = top[sn] (change next(i) to point prev)

⑤ top[sn] = i

while pop()

① get the top index from the topOfStack array (i)

② get the previous of the stack using next[i]

now next[i] becomes top

top[sn] = next[i]

next[i] = freeTop (fill with next available location)

freetop = i

```
class kstacks {
    int *arr, *top, *next;
    int k, freeTop, cap;
    KStacks (int k1, int n){
        k = k1;
        cap = n;
        arr = new int[cap];
        next = new int[cap];
        top = new int[k];
        for (int i = 0; i < k; i++){
            top[i] = -1
        }
        freeTop = 0;
        for (int i=0; i< cap-1; i++){
            next[i] = i+1;
        }
        next[cap-1] = -1;
    }
    void push (sn, x){
        arr[freeTop] = x;
int n   next[freeTop] = top[sn];
        top[sn] = freetop
        freeTop = next[freetop];
        next[freetop] = n;
```

Push Function

```
void push (int x){
    int i = freeTop;
    freeTop = next[i];
    next[i] = top[sn];
    top[sn] = i;
    arr[i] = x;
}
```

pop function

```
int pop(int sn){
    int i = top[sn];
    top[sn] = next[i];
    next[i] = freeTop;
    freeTop = i;
    return arr[i];
}
```

Stock span Problem

I/P → arr[] = {13, 15, 12, 14, 16, 8, 6, 4, 10, 30}

O/P → {1, 2, 1, 2, 5, 1, 1, 1, 4, 10}

span → no of consecutive days with values smaller or same

I/P → arr[] = {10, 20, 30, 40} (increasing order)

O/P → {1, 2, 3, 4}

I/P → {40, 30, 20, 10} (decreasing order)

O/P → {1, 1, 1, 1}

I/P → {30, 20, 25, 28, 27, 29}.

O/P → {1, 1, 2, 3, 1, 5}

Naive Solution

For every element we go to the left side of it and count the number of elements smaller than the curr element

Efficient Solution

span[i] = index of curr Element(i) − (Index of closest greater element on left side).

{60, 10, 20, 40, 35, 30, 50, 70, 65}.

Index →	0	1	2	3	4	5	6	7	8
Formula →	(1)	(1-0)	(2-0)	(3-0)	(4-3)	(5-4)	(6-0)	(7+1)	(8-7)
Ans →	1	1	2	3	1	1	6	8	1

Also span[i] = i+1 (when there is no greater element on left side)

$x_0\ x_1\ x_2\ x_3 \cdots x_i \mid x_{i+1}\ x_{i+2} \cdots x_{n+1}$

We have all the info regarding these element (spans)

For x_{i+1} we have to check the closest greater element at left

- we have to store the chain of elements greater than x_{i+1} on left

For ex

60, 10, 20, 40, 35 | 30, 50, 70, 65

For 30, elements on left

35, 40, 60

There may be 3 possibilities

- element may be smaller than the 1st element of chain
- element must be between two elements of the chain.
- element may be greater than the last element of chain

80, 60, 10, 20, 32, 40, 35 | 30

80, 60, 40, 35

at any point i in the computed part of array, we have to find the element greater than arr[i] on the left to form chain

Let the chain of elements is

$a_n\ a_{n-1} \ldots a_{x+1}\ a_x \cdots a_2\ a_1$

① If x_{n+1} is smaller than a_1, then previous greater element would be simply a_1 (case-1)

② If x_{n+1} is between a_{x+1} _ a_x then previous greater element would be x a_{x+1} (case-2)

③ If x_{n+1} is greater than a_n, then there is no previous greater element, span = i+1 (case-3)

④ Processing starts from right to left (LIFO)

1- example (60, 10, 20, 40, 35, 38, 50 ...

For element 50: 60 40 38 ← chain

Now we have to process '38' first then 40 and then 60.

suppose the element whose span is to be calculated is x

① Now if x < 38 span becomes index of x − index of 38

② else if x < 40, now span becomes index of x − index of 40

and 38 is of no use for use, the chain will be updated to 60, 40, x

③ else if x < 60 span = index of x − index of 60

& chain → 60, x

38
40
60

Example

30, 20, 25, 28, 27, 29.

index → 0 1 2 3 4 5

stack

at i=0

as stack = empty

span = 0+1 (i+1)

= 1

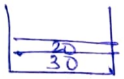
≠ element pushed to stack

at i=1

20 < 30

span = 1 − 0 = 1

20 is pushed in stack



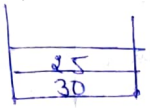
at i=2

25 > 20

20 is poped

~~30~~ 25 < 30

span = 2 − 0 = 2



at i=3

28 > 25

25 poped

28 < 30 span = 3 − 0

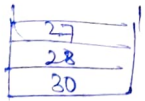
28 pushed

at i=4

27 < 28

span = 4 − 0 = 4

27 pushed



at i=5

29 > 27

27 poped

29 > 28

28 poped

29 < 30 span = 5 − 0 = 5

Implementation

```
void printspan(int arr[], int n){
    stack s;
    s.push(0);   // processing first element
    print(1);
    for(int i=1; i<n; i++){
        while(s.isEmpty()==false &&
              arr[s.top()] <= arr[i]){
            s.pop();
        }
        span = s.isEmpty() ? i+1 : i-s.top();
        print(span);
        s.push(i);
    }
```

popping until arr[i] < arr[s.top()]

calculating span & pushing element

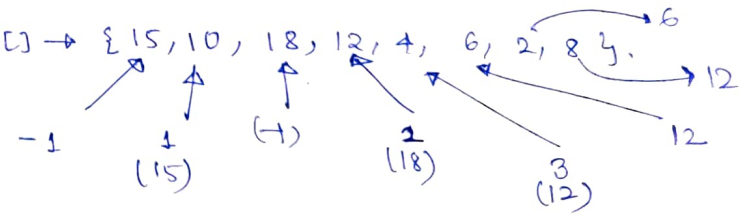
Time complexity → O(n)

Aux-space → O(n) → Ascending
O(1) → Descending
} O(n)

Previous Greater Element

Given an array, we have to find (position-wise) greater element on left. If there is no element (greater) on left then print (-1).

arr[] → {15, 10, 18, 12, 4, 6, 2, 8}.



* now 18 is greater than 12 on left but 12 is pos-wise greater).

* this question is a subpart of previous question

```
void getGreaterPrev(int arr[], int n){
    stack<int> s;
    s.push(0);
    print(-1);
    for(int i=01; i<n; i++){
        while(arr[s.top] < arr[i] &&
              s.isEmpty == = false){
            s.pop();
        }
        if if(s.isEmpty){
            print(-1)
        }
        else {
            print(arr[s.top()])
        }
        s.push(i);
    }
}
```

Next Greater Element

(position wise closest)

I/P : 15, 10, 8, 6, 12, 9, 18}
10, 12, 12, 12, 18, 18, -1}.

Naive solution

for every element, traverse ~~all~~ to right and find the closest greater element

$O(n^2)$

Efficient solution

same approach, just we have to traverse from right

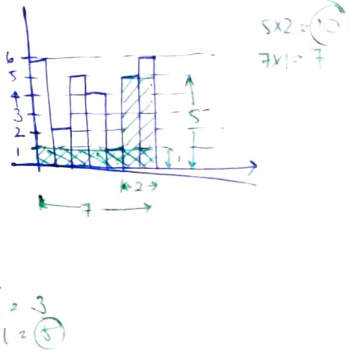
```
void nextGreater(int arr[], int n){
    stack<int> s;
    s.push(arr[n-1]); cout << "-1";
    for(int i= n-2; i>= 0; i--){
        while(s.empty()== false &&
              s.top() <= arr[i]){
            s.pop();
        }
        int nextG = s.empty() ? -1 : s.top();
        cout << nextG;
        s.push(arr[i]);
    }
}
```

- this code produces output in reverse order, we can just store the answers in stack and pop them in the end of function.
* or we can use a vector and reverse it in the end using reverse(v.begin(), v.end());

Largest Rectangular Area in Histogram

I/P → arr → {6, 2, 5, 4, 1, 5, 6}
O/P → 10.

I/P → arr → {2, 5, 1}
O/P → 5



Naive solution

For every bar we calculate the area of rectangle (max-area) having that ~~bar~~ bar as smallest height

area = height × width

arr[i] — to increase this width, we move in left and right to find till we encounter a smaller element

```
int getMaxRes(int arr[], int n){
    int res = 0;
    for(int i=0; i<n; i++){
        int curr = arr[i];
        for(int j= i+1; j<n; j++){
            if(arr[j] >= arr[i]){
                curr += arr[i];
            }
            else break;
        }
        for(int j= i-1; j>=0; j--){
            if(arr[j] >= arr[i])
                curr += arr[i];
            else break;
        }
        res = max(res, curr);
    }
    return res;
}
```

going to check right boundary of rectangle

checking left boundary of rectangle

getting max

Better solution O(n)

We can precompute next & prev smaller element using the same logic (next & prev greater elements).

→ We will push element ~~it~~ in the stack and will pop elements until ~~arr~~ s.top() < arr[i] or s.isEmpty() = true

- if prev smaller does not exist → -1
- if next smaller doesn't exist → n

Steps

① Initialize res = 0

② Find previous smaller & next smaller element for every element

③ For every element arr[i].

```
curr = arr[i].
curr += (i - ps[i] - 1) * arr[i].   // (i-(ps+1))
curr += (ns[i] - i - 1) * arr[i];   // (ns-(i+1))
res = max(res, curr);
```

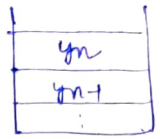
④ return res.

Time complexity → O(3n) → O(n) Space → O(n)

Efficient solution :-

- We want to get the prev and next smaller element in one loop.
- We maintain the stack in such a way so that when we push an element i in the stack the ~~item~~ item below it will be previous greater element.
- When we process the next element, and if the condition ~~sta~~ satisfies and we pop some element, then ~~for~~ for ex.

let sp suppose curr element → x
and stack has elements



then if we pp pop (yn) → for element yn
next smaller = x
prev smaller = y

area of rectangle formed by yn → yn × (index of x − index of yn-1)

PROCESS

① Create a stack s.

② int res = 0;

③ for (int i = 0; i<n; i++){

while (s.empty() == false && arr[s.top()] >= arr[i]){

top = s.pop();

curr = arr[top] * (s.empty ? i : i − s.top()−1);
(height; getting width)

res = max(res, curr);

}

s.push(i); }

while (s.empty() == false){

top = s.pop();

curr = arr[top] * (s.empty ? n : (n − s.top()−1));

res = max(res, curr);

}. return res;

(*) This while loop is for element which do not have any next smaller element; and this is the reason why they never pop out from the stack.

For these element, we consider the previous smaller and find the area

curr = (n) − s.top()−1 * arr[top];

get all the elements on right side — previous smaller

Largest Rectangular submatrix with all 1's

I/P: mat[][] = {{0, 1, 1, 0},
{1, 1, 1, 1},
{1, 1, 1, 1},
{1, 1, 0, 0}}

① → 2×3 = 6
② → 2×4 = 8 ✓
③ → 2×3 = 6

O/P = 8

I/P: mat[][] = {{0, 1, 1},
{1, 1, 1},
{0, 1, 1}}

2×3 = 6 ✓
1×3 = 3
2×1 = 2

I/P = mat[],[] = {{0,0},
{0, 0}}.

0×0 = 0

we can use the largest area of histogram problem to solve this.

we can consider, every row and calculate the (as a base) max area formed by bars of all 1's

For ex

$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} \rightarrow \begin{matrix} [0,1,1,0] \\ [1,2,2,1] \\ [2,3,3,2] \\ [3,4,0,0] \end{matrix}$$

If we start by the last row, we have to compute the height ~~for~~ each column, that again calls for an $R \times C$ loop.

Efficient solution is to start by the top and then pass the 1d row array to the function while calculating the longest histogram for 2nd row.

We can have something like this →

① longestHistogram (Array) → gives the value of max. area formed by arr val.

② sumRows (vector a, vector b) {

```
if b[i] = 0
   a[i] = 0
else
   a[i]++;
}
```

③ Call longest histogram for arr[i] & then a[i+1] = sumRows (arr[i], arr[i+1])

Problem — 85 (Leetcode)

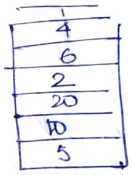
Design a stack that supports getMin().

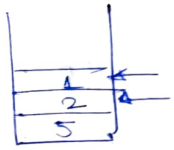
I/P → push(20), push(10), getMin(), push(5), getMin()

O/P → 10, 5

We maintain an auxiliary array where we will push only those elements who is minimum

Ex:- 5, 10, 20, 2, 6, 4, 1





1 is smaller.
2 is smaller than 5

push() →

```
if (s.empty()) {
    s.push(x);
    as.push(x);
}
```

```
s.push(x)
if (as.top >= s.top())
    as.push(x);
```

pop() →

```
if (s.top() == as.top())
    as.pop();
s.pop();
```

aux space – O(n)
time comp – O(1)

* Having O(1) Aux space

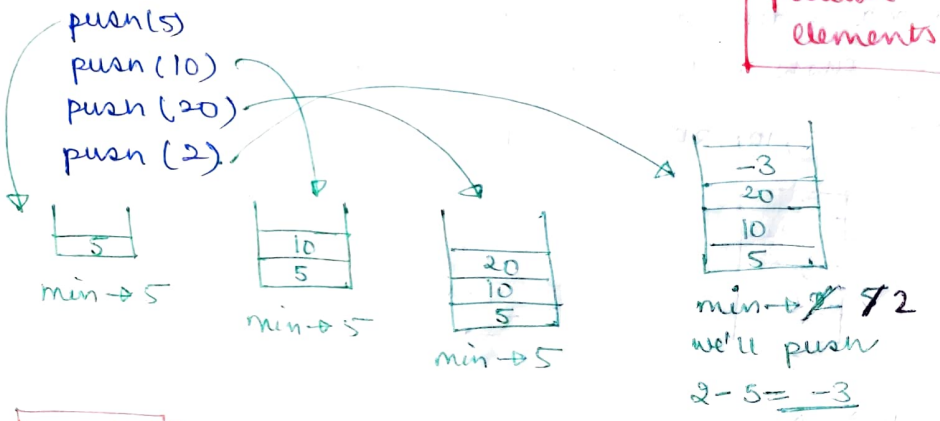
- The task is we have to store prev minimum elements and the curr minimum without using any data structure
- We have to store 2 values in 1 space i.e 1 variable.

if

we will use a variable min → stores the curr min elem.

whenever we push(x) and x < min, we will store the ~~min~~ x - min in stack and min becomes x

Only positive elements




```
push(x) {
    if (s.empty()){
        min = x;
        s.push(x);
    }
    else if (x <= min) {
        s.push(x - min);
        min = x;
    }
    else {
        s.push(x);
    }
```

```
int getmin(){
    return min;
}
```

```
int pop(){
    t = s.top(); s.pop();
    if (t <= 0){
        res = min
        min = min - t;
        return res
    }.
    else {
        return t;
    }.
}.
```

```
int ipeek(){
    t = s.peek();
    return (t<=0)?
                min : t
}.
```

Second Method - Handling Negatives

- Initially when we were storing, ~~min~~ the special value is s.top() ≤ ~~0~~ 0

Here special value will be s.top ≤ min

We know that x - min ≤ 0

2x - min = x + (x - min)

≤ x

special case → 2x - min → element pushed ≤ x

Also we are pushing 2x x - min

where x → new min

min → prev min

2x min - prev_min = top of stack

prev_min = 2x min - top_stack

```
Void push(int x){
    if(s.isEmpty()){
        min = x;
        s.push(x);
    }
    else if(x <= min) {
        s.push(2*x - min);
        min = x; }
    else
        s.push(x);
}
```

```
void pop(~~int~~){
    t = s.pop();
    if (t <= min){
        res = min;
        min = 2* min
              -t;
        return res;
    }
    else
        return t;
}.
```

Infix, Prefix and Postfix :

- Infix: $x+y$
- Prefix: $+xy$
- Postfix: $xy+$

Operators	Associativity
$\wedge$	R to L
$*, /$	L to R
$+, -$	L to R

↑ precedence

closing bracket (bottomost priority)

Steps to convert infix to postfix

1. Parenthesis expression using associativity and precedence rules
2. convert the innermost to postfix

Ex1 $x+y*z \rightarrow x+(y*z) \rightarrow x+(yz*) \rightarrow xyz*+$

Ex-2 $(x+y)*z \rightarrow ((x+y)*z) \rightarrow ((xy+)*z) \rightarrow xy+z*$

Infix to Postfix

~~operators~~

$\wedge$

$*$

I/P → $a+b*c$

O/P → $bc*a+$

I/P → $(a+b)*c$

$ab+c*$

prefix & postfix are better because

- Do not require parenthesis, precedence rules, and associativity rules
- can be evaluated by writing a program that traverses the given expression exactly once

Naive Approach

→ Fully parenthesise the expression

→ And then start with innermost expression

1. $a \wedge b \wedge c \rightarrow (a \wedge (b \wedge c)) \rightarrow (a \wedge (bc\wedge)) \rightarrow abc\wedge\wedge$
2. $((a+b)*(c+d)) \rightarrow (ab+)*(cd+) \rightarrow ab+cd+*$
3. $a+b*(c-d) \rightarrow a+(b*(c-d)) \rightarrow a+(b*(cd-)) \rightarrow a+(bcd-*) \rightarrow abcd-*+$
4. $a+b*c/d+e \rightarrow a+((b*c)/d)+e \rightarrow a+((bc*)/d)+e \rightarrow a+(bc*d/)+e \rightarrow (abc*d/+)+e \rightarrow abc*d/+e+$

Efficient solution

Steps

1. create an empty stack S.
2. For every character x from left to right
3. If x is
   - (a) operand : print it
   - (b) left parenthesis :- push to the stack
   - (c) right parenthesis :- pop elements from the stack until left par. is found.

(d) operator – (i) If S is empty, push x to S
(ii) else compare x with S.top()

- Higher precendence than S.top() → push to S (i)
- Lower precedence than S.top() → Output until higher precedence operator is found. Push x to S (ii)
- Equal priced → use associativity
  - If associativity is L to R (ii)
  - R to L (i)

Input : a+b*c

Input	Stack	Result (Postfix)
a		a
+	+	a
b	+	ab
*	* +	ab
c	* +	abc

pop everything from stack → abc * +

Example-2 (a+b) * c

Input symbol	Stack	Result
(	(	
a	(	a
+	(+	a
b	(+	ab
)		ab+
*	*	ab +
c	*	ab+c
pop the remaining stack		ab+c*

Example-3 (a * b) / c

Input	Stack	Result
(	(	
a	(	a
*	(*	a
b	(*	ab
)		ab*
/	/	ab*
c	/	ab* c
		ab * c /

Ex-4 a * b / c

Input	Stack	Result
a		a
*	*	a
b	*	ab
/	/	ab*
c	/	ab*c
		ab*c/

Example-5 - a + b / c - d * e

Input	Stack	Output
a	[ ]	a
+	[+]	a
b	[+]	ab
/	[+ /]	ab
c	[+ /]	abc
-	[-]	abc/+
d	[-]	abc/+d
*	[- *]	abc/+d
e	[- *]	abc/+de
pop everything		abc/+de*-

Ex Evaluation of Postfix

I/P → 10 2 * 3 +
O/P → 23

I/P → 10 2 + 3 *
O/P → 36.

Algorithm

① operand → push in stack
② Operator → pop 2 elements and apply that operator on 2 elements & push result to stack

Ans → s.top() at the end of thing

Infix to Prefix Conversion

x+y → +xy

I/P → x + y * z
O/P → x + (y * z) ⇒ x + (* yz)
⇒ + x * yz

I/P → (x+y) * z
O/P → (+xy) * z
→ * +xyz.

I/P → 4 ^ 5 ^ 3
O/P → 4 ^ (5 ^ 3) ⇒ 4 ^ (^ 53)
⇒ ^ 4 ^ 53.

I/P → x + (y * (z - w)
O/P → x + (y * (-zw))
x + (* y - zw)
+ x * y - zw.

I/P → x + y * z / w + u
x + ((y * z) / w) + u
(x + ((* yz) / w)) + u
x + (/ * yz w) + u
(+ x / * yz w) + u
+ + x / * yz w u

using stacks

(Traverse Right to Left)

① create a stack, s
② create a string 'prefix'
③ For every character s[i] = x
if x is (a) operand → push to prefix
(b) Right parenthesis → push to prefix
(c) left parenthesis → pop from stack until right part. is found.

(d) operator -
(i) If s is empty, then push x to s else compare it with s.top

(ii) higher precedence - push to stack

(iii) lower precedence - ~~push~~ pop elements from stack until high priority element found.
→ Push x to stack.

(iv) Equal Precedence → If operator is L to R associative.
↓
if R to L associative
incoming has lower precedence

incoming has higher precedence

(v) pop all the remaining elements
(vi) reverse the prefix string.

Input → x + y * z

Input	stack
x	
*	*
y	*
* +	+
x	+

pop all the elements from stack

Prefix (Reverse)

x
x
xy
~~xyx~~ xy*
~~xy*x~~ xy*x
~~xy*x+~~
xy*x+
↓
Reverse
↓
+x*yz

Input - (x+y)*z

Input symbol	stack
~~(~~ z	
*	*
)	*)
y	*)
+	*) +
x	*) +
(	*

pop everything out

Right to left

Prefix (Reverse)

z
z
z
zy
zy
zyx
zyx +
~~zyx+*~~
↓
Reverse
↓
*+xyz

Input - $x \wedge_2 y \wedge_1 z$

Input	stack
z	
$\wedge_1$	$\wedge_1$
y	$\wedge_1$
$\wedge_2$	$\wedge_2$
x	$\wedge_1$

As its it is Right associative, $\wedge_1$ has higher precedence

Prefix

z
z
zy
zy∧
zy∧x
zy∧x∧
↓
∧x∧yz

Infix → x + y / z − w * u

Input	Stack	prefix
u	[ ]	u
*	[*]	u
w	[*]	uw
−	[−]	uw*
z	[−]	uw*z
/	[− /]	uw*z
y	[− /]	uw*zy
+	[− +]	uw*zy/
x	[− +]	uw*zy/x

pop()

uw*zy/x+−

↓ Reverse

−+x/yz*wu

Evaluation of Prefix expression

I/P − + * 10 2 3

O/P → 23

- Scan ~~to~~ from right to left
- Whenever operator appears pop last two elements from stack ~~push~~ & push the op1 operator op2 to the stack.
- s.top() → Result