[Design a stack such that getMinium() should be O(1) time and O(1) space](https://interviewprep.appliedcourse.com/lecture/2/interview-preparation-course/893/design-a-stack-such-that-getminium-should-be-o1-time-and-o1-space/18/module-4-problem-solving)

class MinStack:

def \_\_init\_\_(self):

"""

initialize your data structure here.

"""

self.stack = []

self.minstack = []

def push(self, x: int) -> None:

self.stack.append(x)

if not self.minstack or x <= self.minstack[-1]:

self.minstack.append(x)

def pop(self) -> None:

if self.minstack and self.stack[-1] == self.minstack[-1]:

self.minstack.pop()

if self.stack:

self.stack.pop()

def top(self) -> int:

if self.stack:

return self.stack[-1]

return 2\*\*64-1

def getMin(self) -> int:

if self.minstack:

return self.minstack[-1]

return 2\*\*64-1

Code:- 2:

class MinStack:

def \_\_init\_\_(self):

"""

initialize your data structure here.

"""

self.stack = []

def push(self, x: int) -> None:

if not self.stack:

self.stack.append((x, x))

else:

\_, minnum = self.stack[-1]

if x >= minnum:

self.stack.append((x, minnum))

else:

self.stack.append((x, x))

def pop(self) -> None:

if self.stack:

self.stack.pop()

def top(self) -> int:

if self.stack:

return self.stack[-1][0]

return 2\*\*64-1

def getMin(self) -> int:

if self.stack:

return self.stack[-1][1]

return 2\*\*64-1

Next Greater Element

Given an array, print the Next Greater Element (NGE) for every element. The Next greater Element for an element x is the first greater element on the right side of x in array. Elements for which no greater element exist, consider next greater element as -1.

# Function to print element and NGE pair for all elements of list

def printNGE(arr):

    for i in range(0, len(arr), 1):

        next = -1

        for j in range(i+1, len(arr), 1):

            if arr[i] < arr[j]:

                next = arr[j]

                break

        print(str(arr[i]) + " -- " + str(next))

# Driver program to test above function

arr = [11,13,21,3]

printNGE(arr)

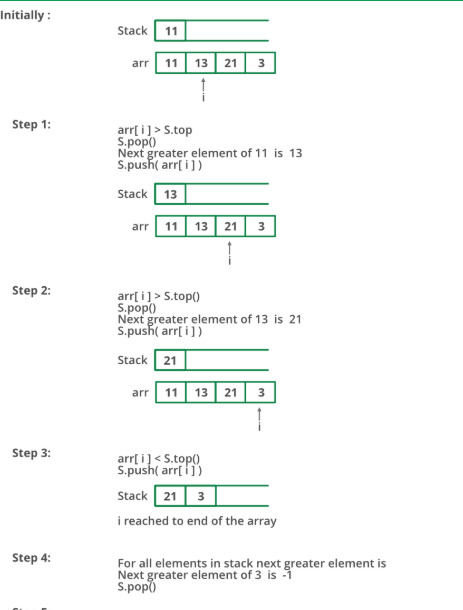
**Output:**

11 -- 13

13 -- 21

21 -- -1

3 -- -1



# Python program to print next greater element using stack

# Stack Functions to be used by printNGE()

def createStack():

    stack = []

    return stack

def isEmpty(stack):

    return len(stack) == 0

def push(stack, x):

    stack.append(x)

def pop(stack):

    if isEmpty(stack):

        print("Error : stack underflow")

    else:

        return stack.pop()

'''prints element and NGE pair for all elements of

   arr[] '''

def printNGE(arr):

    s = createStack()

    element = 0

    next = 0

    # push the first element to stack

    push(s, arr[0])

    # iterate for rest of the elements

    for i in range(1, len(arr), 1):

        next = arr[i]

        if isEmpty(s) == False:

            # if stack is not empty, then pop an element from stack

            element = pop(s)

            '''If the popped element is smaller than next, then

                a) print the pair

                b) keep popping while elements are smaller and

                   stack is not empty '''

            while element < next :

                print(str(element)+ " -- " + str(next))

                if isEmpty(s) == True :

                    break

                element = pop(s)

            '''If element is greater than next, then push

               the element back '''

            if  element > next:

                push(s, element)

        '''push next to stack so that we can find

           next greater for it '''

        push(s, next)

    '''After iterating over the loop, the remaining

       elements in stack do not have the next greater

       element, so print -1 for them '''

    while isEmpty(s) == False:

            element = pop(s)

            next = -1

            print(str(element) + " -- " + str(next))

# Driver program to test above functions

arr = [11, 13, 21, 3]

printNGE(arr)

# This code is contributed by Sunny Karira

# Queue using Stacks

*enQueue(q, x):*

* *While stack1 is not empty, push everything from stack1 to stack2.*
* *Push x to stack1 (assuming size of stacks is unlimited).*
* *Push everything back to stack1.*

*Here time complexity will be O(n)*

*deQueue(q):*

* *If stack1 is empty then error*
* *Pop an item from stack1 and return it*

*Here time complexity will be O(1)*

# Python3 program to implement Queue using

# two stacks with costly enQueue()

class Queue:

    def \_\_init\_\_(self):

        self.s1 = []

        self.s2 = []

    def enQueue(self, x):

        # Move all elements from s1 to s2

        while len(self.s1) != 0:

            self.s2.append(self.s1[-1])

            self.s1.pop()

        # Push item into self.s1

        self.s1.append(x)

        # Push everything back to s1

        while len(self.s2) != 0:

            self.s1.append(self.s2[-1])

            self.s2.pop()

    # Dequeue an item from the queue

    def deQueue(self):

            # if first stack is empty

        if len(self.s1) == 0:

            print("Q is Empty")

        # Return top of self.s1

        x = self.s1[-1]

        self.s1.pop()

        return x

# Driver code

if \_\_name\_\_ == '\_\_main\_\_':

    q = Queue()

    q.enQueue(1)

    q.enQueue(2)

    q.enQueue(3)

    print(q.deQueue())

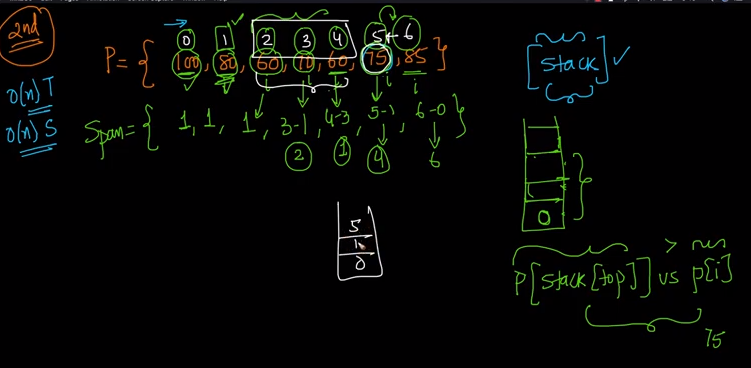
    print(q.deQueue())

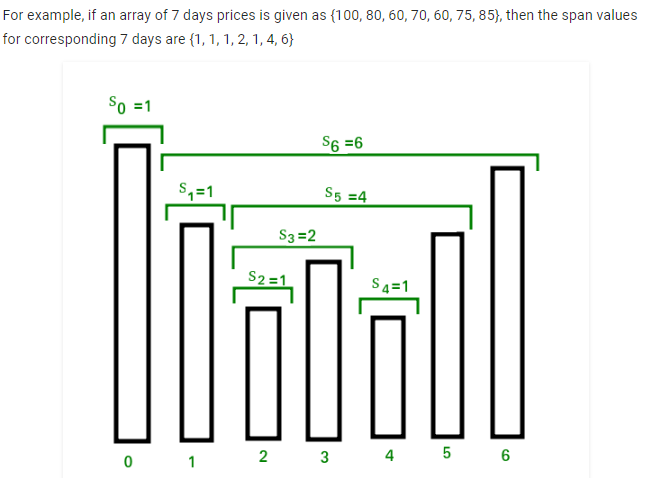
    print(q.deQueue())

# This code is contributed by PranchalK

# The Stock Span Problem

[The stock span problem](http://en.wikipedia.org/wiki/Stack_(abstract_data_type)#The_Stock_Span_Problem) is a financial problem where we have a series of n daily price quotes for a stock and we need to calculate span of stock’s price for all n days.  
The span Si of the stock’s price on a given day i is defined as the maximum number of consecutive days just before the given day, for which the price of the stock on the current day is less than or equal to its price on the given day.  
For example, if an array of 7 days prices is given as {100, 80, 60, 70, 60, 75, 85}, then the span values for corresponding 7 days are {1, 1, 1, 2, 1, 4, 6}





Inefficient method:-

def calculateSpan(price, n, S):

    # Span value of first day is always 1

    S[0] = 1

    # Calculate span value of remaining days by linearly

    # checking previous days

    for i in range(1, n, 1):

        S[i] = 1   # Initialize span value

        # Traverse left while the next element on left is

        # smaller than price[i]

        j = i - 1

        while (j>= 0) and (price[i] >= price[j]) :

                       S[i] += 1

                       j -= 1

Efficient One:--

def span(rates):

stockspan = []

stack = [] #creating an empty stack

for i in range(len(rates)):

#Base case

if i == 0:

stockspan.append(1)

stack.append(0)

continue

#Pop elements out of stack untill either: 1) The stack gets empty

#or 2) the rate at stack top turns out to be larger than the rate

#at the current element

while rates[i] > rates[stack[-1]]:

if len(stack) == 0:

break

stack.pop()

#set the stockspan values.

if len(stack)>0:

stockspan.append(i - stack[-1])

else:

stockspan.append(i + 1)

stack.append(i)

return stockspan

def main():

rates = [31, 27, 14, 21, 30, 22]

stockspan = span(rates)

print(stockspan)

main()

# Implement Stack using Queues

**Method 1 (By making push operation costly)**  
This method makes sure that newly entered element is always at the front of ‘q1’, so that pop operation just dequeues from ‘q1’. ‘q2’ is used to put every new element at front of ‘q1’.

1. **push(s, x)** operation’s step are described below:
   * Enqueue x to q2
   * One by one dequeue everything from q1 and enqueue to q2.
   * Swap the names of q1 and q2
2. **pop(s)** operation’s function are described below:
   * Dequeue an item from q1 and return it.

|  |
| --- |
| 1. # Program to implement a stack using 2. # two queue 3. from queue import Queue 5. class Stack: 7. def \_\_init\_\_(self): 9. # Two inbuilt queues 10. self.q1 = Queue() 11. self.q2 = Queue() 13. # To maintain current number 14. # of elements 15. self.curr\_size = 0 17. def push(self, x): 18. self.curr\_size += 1 20. # Push x first in empty q2 21. self.q2.put(x) 23. # Push all the remaining 24. # elements in q1 to q2. 25. while (not self.q1.empty()): 26. self.q2.put(self.q1.queue[0]) 27. self.q1.get() 29. # swap the names of two queues 30. self.q = self.q1 31. self.q1 = self.q2 32. self.q2 = self.q 34. def pop(self): 36. # if no elements are there in q1 37. if (self.q1.empty()): 38. return 39. self.q1.get() 40. self.curr\_size -= 1 42. def top(self): 43. if (self.q1.empty()): 44. return -1 45. return self.q1.queue[0] 47. def size(self): 48. return self.curr\_size 50. # Driver Code 51. if \_\_name\_\_ == '\_\_main\_\_': 52. s = Stack() 53. s.push(1) 54. s.push(2) 55. s.push(3) 57. print("current size: ", s.size()) 58. print(s.top()) 59. s.pop() 60. print(s.top()) 61. s.pop() 62. print(s.top()) 64. print("current size: ", s.size()) 66. # This code is contributed by PranchalK |

**Output :**

current size: 3

3

2

1

current size: 1

# Implement two stacks in an array

**Method 2 (A space efficient implementation)**  
This method efficiently utilizes the available space. It doesn’t cause an overflow if there is space available in arr[]. The idea is to start two stacks from two extreme corners of arr[]. stack1 starts from the leftmost element, the first element in stack1 is pushed at index 0. The stack2 starts from the rightmost corner, the first element in stack2 is pushed at index (n-1). Both stacks grow (or shrink) in opposite direction. To check for overflow, all we need to check is for space between top elements of both stacks. This check is highlighted in the below code.

# Python Script to Implement two stacks in a list

class twoStacks:

    def \_\_init\_\_(self, n):     #constructor

        self.size = n

        self.arr = [None] \* n

        self.top1 = -1

        self.top2 = self.size

    # Method to push an element x to stack1

    def push1(self, x):

        # There is at least one empty space for new element

        if self.top1 < self.top2 - 1 :

            self.top1 = self.top1 + 1

            self.arr[self.top1] = x

        else:

            print("Stack Overflow ")

            exit(1)

    # Method to push an element x to stack2

    def push2(self, x):

        # There is at least one empty space for new element

        if self.top1 < self.top2 - 1:

            self.top2 = self.top2 - 1

            self.arr[self.top2] = x

        else :

           print("Stack Overflow ")

           exit(1)

    # Method to pop an element from first stack

    def pop1(self):

        if self.top1 >= 0:

            x = self.arr[self.top1]

            self.top1 = self.top1 -1

            return x

        else:

            print("Stack Underflow ")

            exit(1)

    # Method to pop an element from second stack

    def pop2(self):

        if self.top2 < self.size:

            x = self.arr[self.top2]

            self.top2 = self.top2 + 1

            return x

        else:

            print("Stack Underflow ")

            exit()

# Driver program to test twoStacks class

ts = twoStacks(5)

ts.push1(5)

ts.push2(10)

ts.push2(15)

ts.push1(11)

ts.push2(7)

print("Popped element from stack1 is " + str(ts.pop1()))

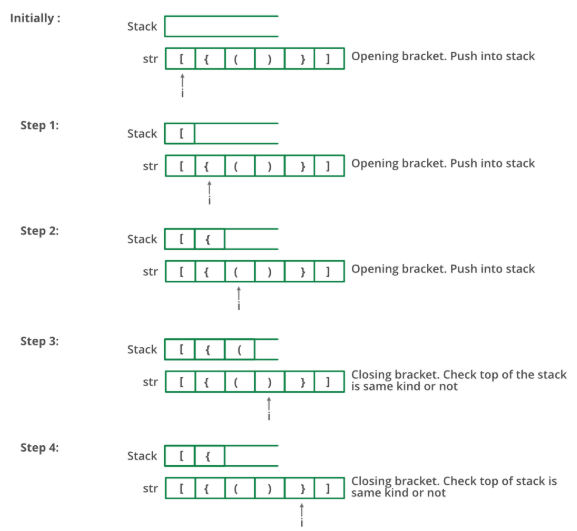
ts.push2(40)

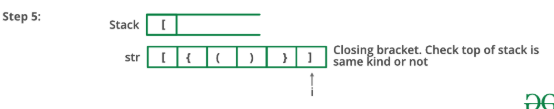
print("Popped element from stack2 is " + str(ts.pop2()))

# Check for balanced parentheses in an expression

***Input****: exp = “[()]{}{[()()]()}”****Output****: Balanced*

***Input****: exp = “[(])”****Output****: Not Balanced*





# Python3 program to check for

# balanced parenthesis.

# function to check if

# paranthesis are balanced

def areParanthesisBalanced(expr) :

    s = [];

    # Traversing the Expression

    for i in range(len(expr)) :

        if (expr[i] == '(' or

            expr[i] == '[' or expr[i] == '{') :

            # Push the element in the stack

            s.append(expr[i]);

            continue;

        # IF current character is not opening

        # bracket, then it must be closing.

        # So stack cannot be empty at this point.

        if (len(s) == 0) :

            return False;

        if expr[i] == ')' :

            # Store the top element in a

            x = s.pop();

            if (x == '{' or x == '[') :

                return False;

        elif expr[i] == '}':

            # Store the top element in b

            x = s.pop();

            if (x == '(' or x == '[') :

                return False;

        elif x == ']':

            # Store the top element in c

            x = s.pop();

            if (x =='(' or x == '{') :

                return False;

    # Check Empty Stack

    if len(s) :

        return False

    else :

        return True

# Driver Code

if \_\_name\_\_ == "\_\_main\_\_" :

    expr = "{()}[]";

    if (areParanthesisBalanced(expr)) :

        print("Balanced");

    else :

        print("Not Balanced");

# The Celebrity Problem

In a party of N people, only one person is known to everyone. Such a person ***may be present*** in the party, if yes, (s)he doesn’t know anyone in the party. We can only ask questions like “***does A know B?*** “. Find the stranger (celebrity) in minimum number of questions.

We can describe the problem input as an array of numbers/characters representing persons in the party. We also have a hypothetical function HaveAcquaintance(A, B) which returns true if A knows B, false otherwise. How can we solve the problem.

## Solution

* Use the fact that the celebrity knows no one else, and everyone must know the celebrity
* Compare adjacent elements in the array and determine who among the two can be a celebrity
  + If a knows b, a cannot be celebrity, b is a probable celebrity
  + Else if b knows a, then b cannot be the celebrity, a is a probably celebrity
* Repeat above until we have examined all elements in the array

## Code

def identify\_celebrity(arr)

i = 1

celebrity = arr[0]

while (i < arr.size)

celebrity = arr[i] if knows(celebrity, arr[i])

i += 1

end

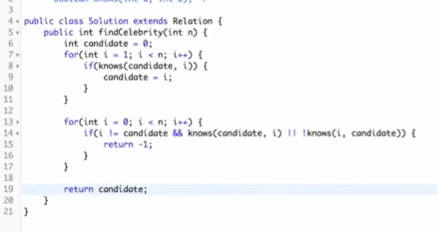
celebrity

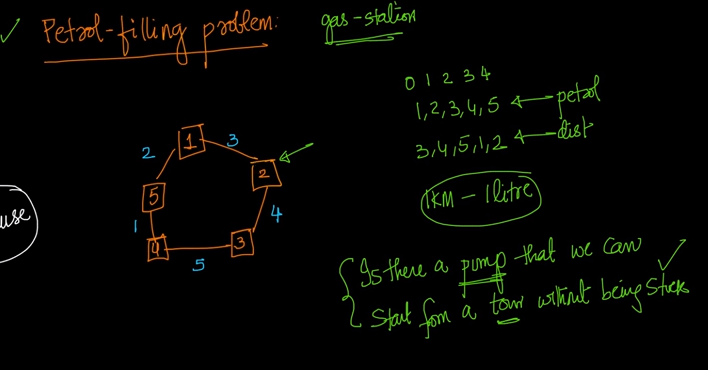
end

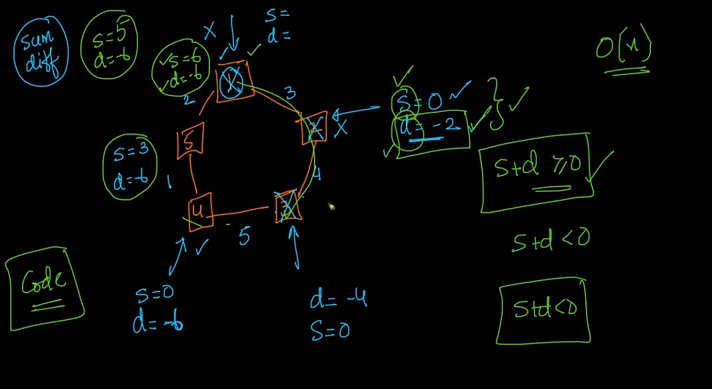
def knows(a, b)

// Returns true if a knows b, else false

end







Which petrol pump to start:=

Easiest approach:

N=input()

sumi=0;

maxi=0;

j=0;

for i in range(N):

a,b=raw\_input().split()

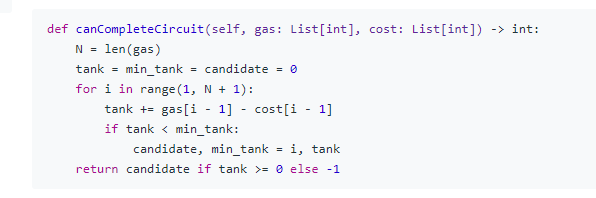
sumi+=int(a)-int(b)

if(sumi<0):

sumi=0

j=i+1

print(j)





**Reverse a stack using recursion**

# Python program to reverse a

# stack using recursion

# Below is a recursive function

# that inserts an element

# at the bottom of a stack.

def insertAtBottom(stack, item):

    if isEmpty(stack):

        push(stack, item)

    else:

        temp = pop(stack)

        insertAtBottom(stack, item)

        push(stack, temp)

# Below is the function that

# reverses the given stack

# using insertAtBottom()

def reverse(stack):

if not isEmpty(stack):

        temp = pop(stack)

        reverse(stack)

        insertAtBottom(stack, temp)

# Below is a complete running

# program for testing above

# functions.

# Function to create a stack.

# It initializes size of stack

# as 0

def createStack():

    stack = []

    return stack

# Function to check if

# the stack is empty

def isEmpty( stack ):

    return len(stack) == 0

# Function to push an

# item to stack

def push( stack, item ):

    stack.append( item )

# Function to pop an

# item from stack

def pop( stack ):

    # If stack is empty

    # then error

    if(isEmpty( stack )):

        print("Stack Underflow ")

        exit(1)

    return stack.pop()

# Function to print the stack

def prints(stack):

    for i in range(len(stack)-1, -1, -1):

        print(stack[i], end = ' ')

    print()

# Driver Code

stack = createStack()

push( stack, str(4) )

push( stack, str(3) )

push( stack, str(2) )

push( stack, str(1) )

print("Original Stack ")

prints(stack)

reverse(stack)

print("Reversed Stack ")

prints(stack)

**Largest Rectangle in Histogram**

Given *n* non-negative integers representing the histogram's bar height where the width of each bar is 1, find the area of largest rectangle in the histogram.

  
Above is a histogram where width of each bar is 1, given height = [2,1,5,6,2,3].

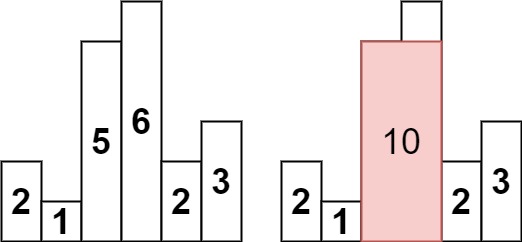
  
The largest rectangle is shown in the shaded area, which has area = 10 unit.

**Example:**

**Input:** [2,1,5,6,2,3]

**Output:** 10

**Example 1:**

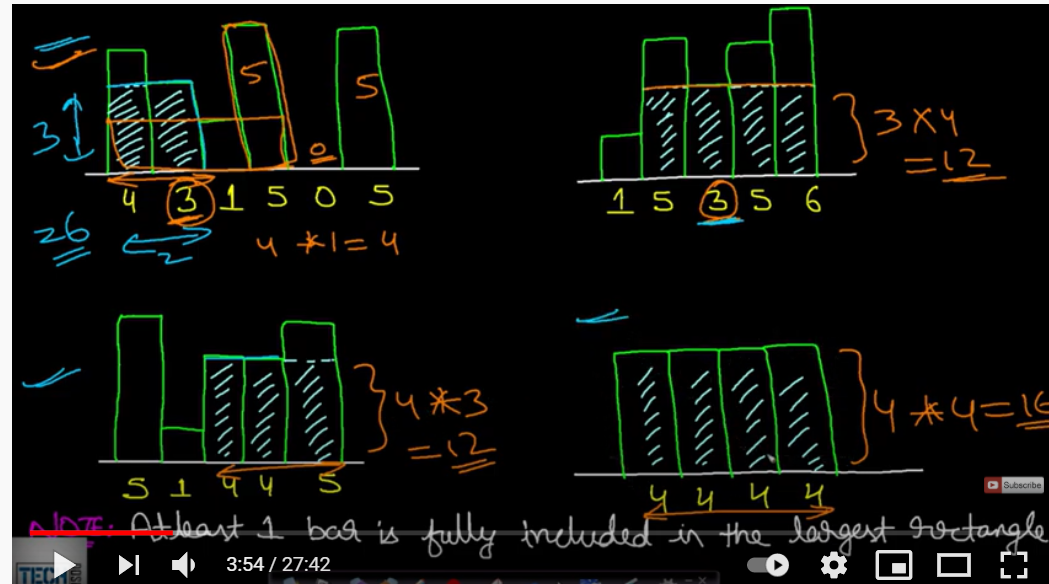


**Input:** heights = [2,1,5,6,2,3]

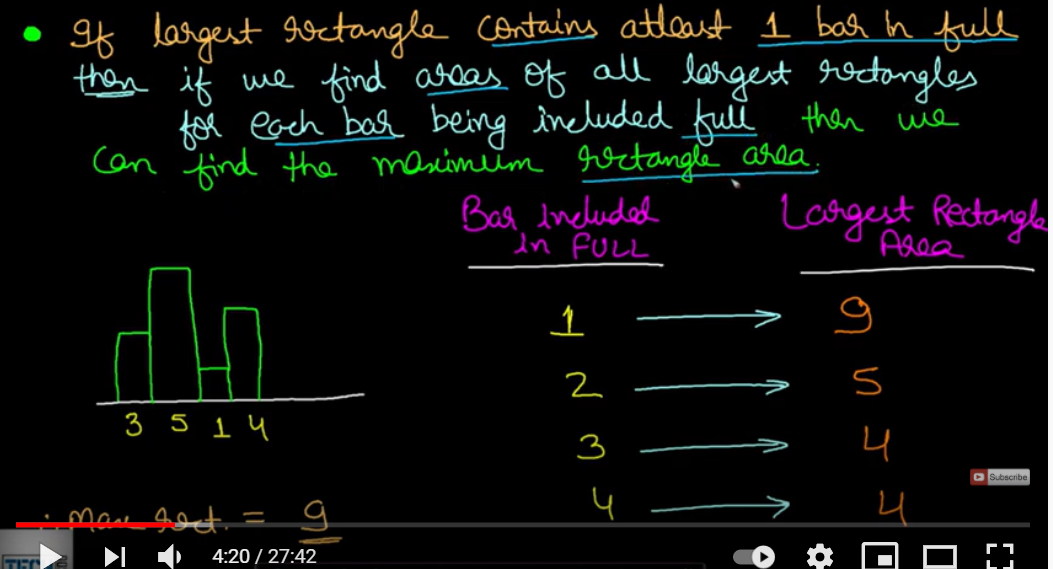
**Output:** 10

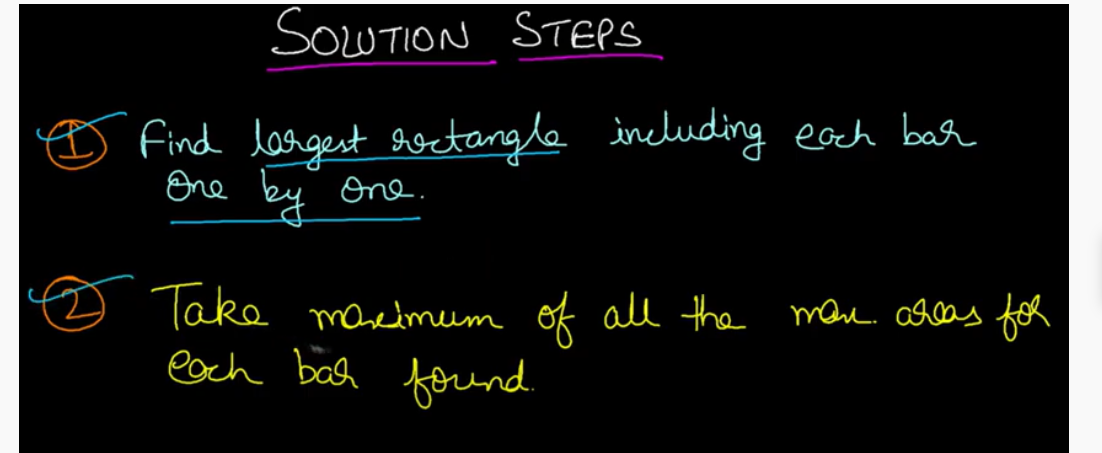
**Explanation:** The above is a histogram where width of each bar is 1.

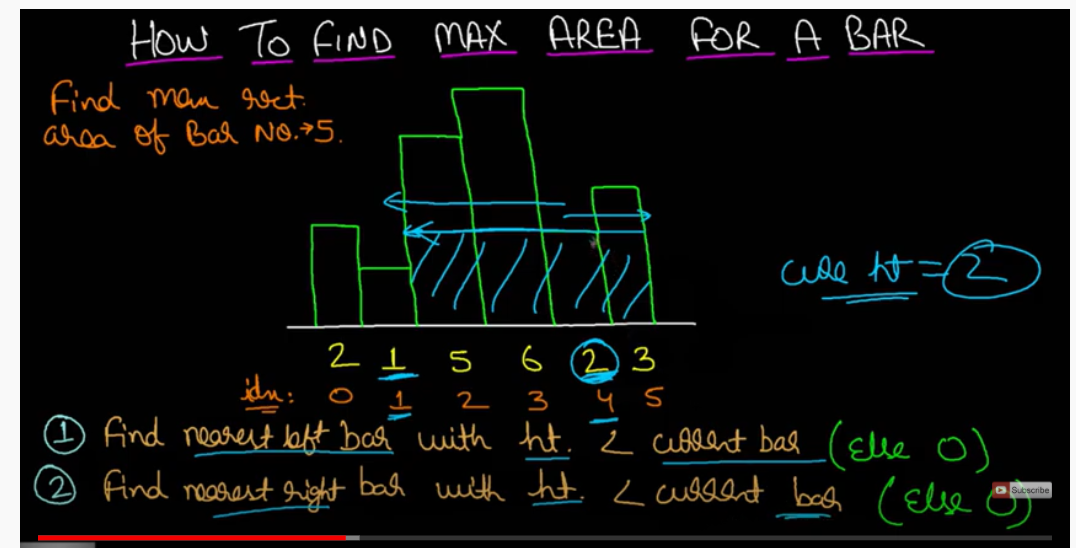
The largest rectangle is shown in the red area, which has an area = 10 units.

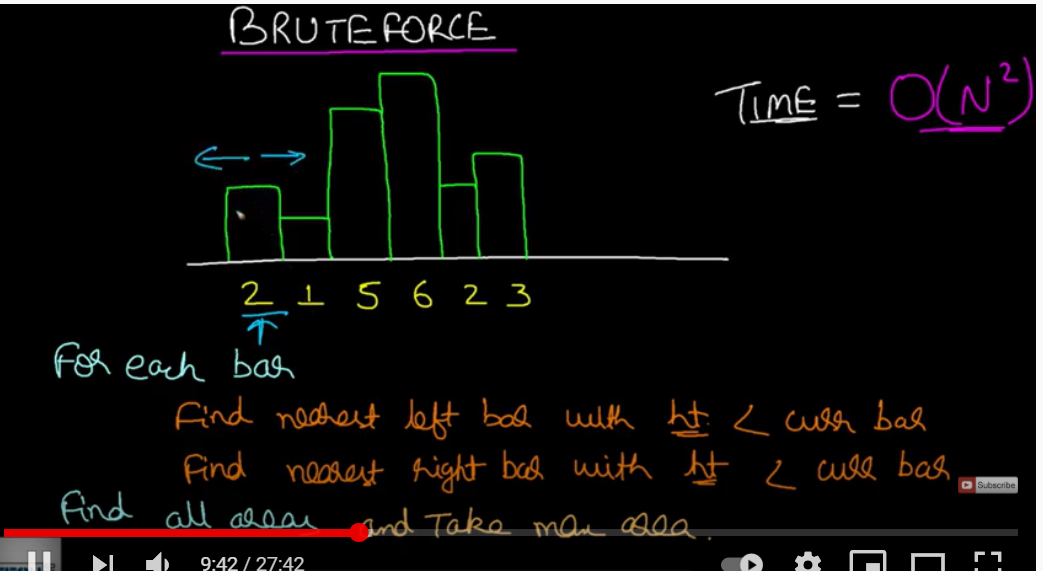


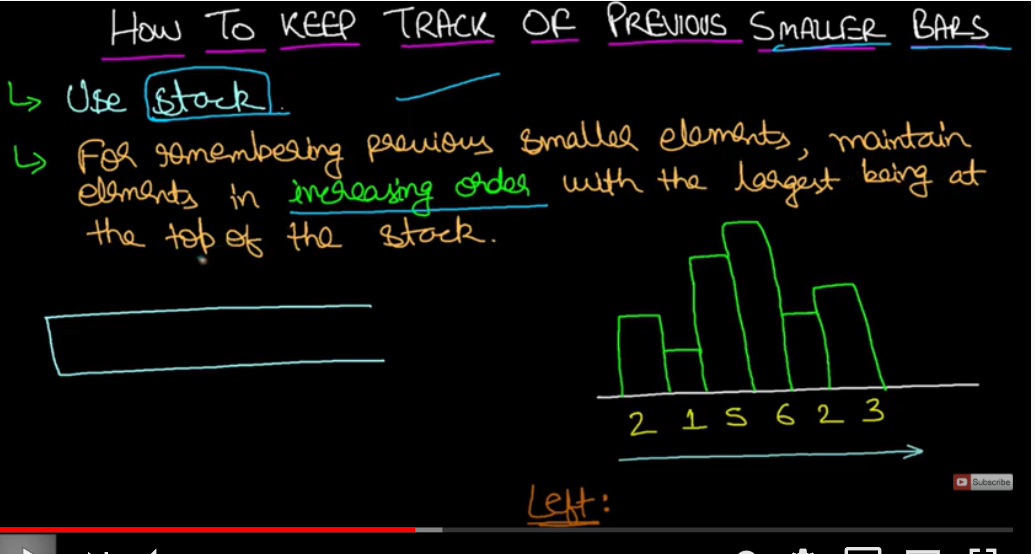
There will be one bar which will be fully included in the largest rectangle.s

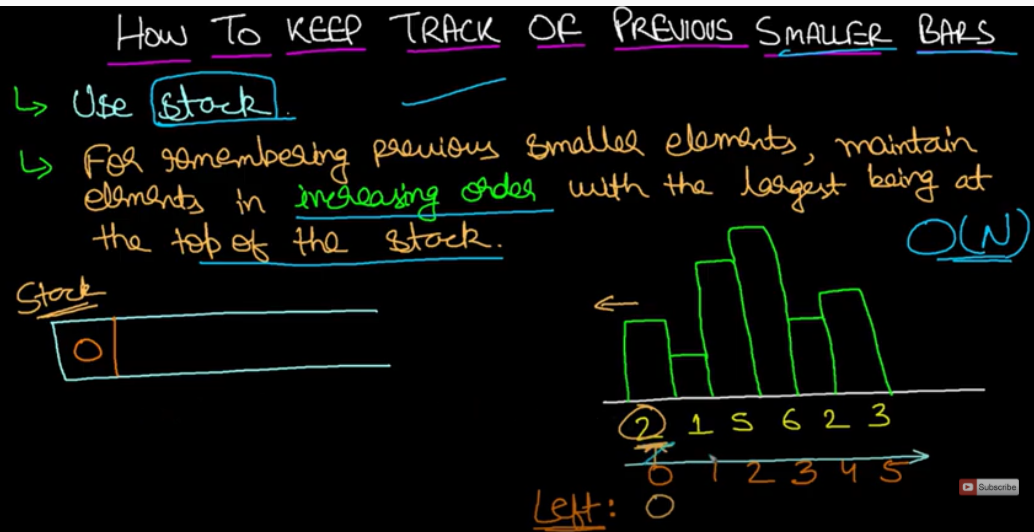
ss



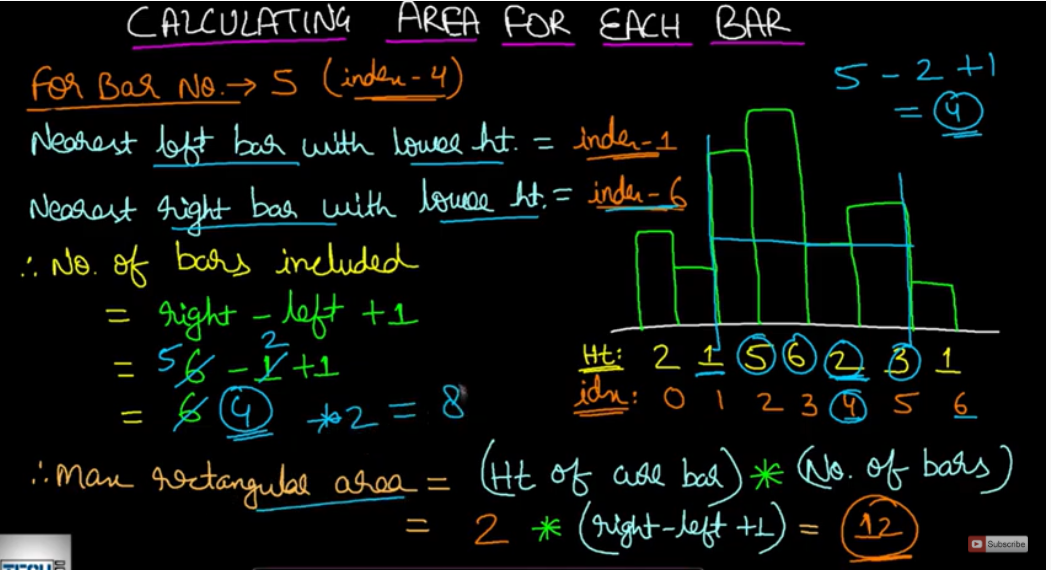
s

s





We will push the index 0 to the stack.



**Key Insights:**

For each index, the largest rectangle of heights h[i] that passes  
through the index will have a left bound at the first height lower  
than h[i] to the left and a right bound at the first height lower than  
h[i] to the right.

**Approach:**

For each index:

1. Find the index of the first shorter building that is to the left
2. Find the index of the first shorter building that is to the right
3. The rectangle area is heights[i] \* (right[i] - left[i] - 1)

**Implementation:**

Use a stack approach to find the index of the first building shorter  
than heights[i] to the right of index i.

Then reverse heights and repeat the process to find the first  
building shorter than heights[i] to the left of i.

def largestRectangleArea(self, heights: List[int]) -> int:

if not heights: return 0

n = len(heights)

# 1. Index of the first building shorter than heights[i] to the right of i

right = [n] \* n

stack = []

for i, h in enumerate(heights):

while stack and h < stack[-1][1]:

right[stack.pop()[0]] = i

stack.append((i, h))

# 2. Index of the first building shorter than heights[i] to the left of i

left = [-1] \* n

stack = []

for i in range(n - 1, -1, -1):

h = heights[i]

while stack and h < stack[-1][1]:

left[stack.pop()[0]] = i

stack.append((i, h))

area = [heights[i] \* (right[i] - left[i] - 1) for i in range(n)]

return max(area)