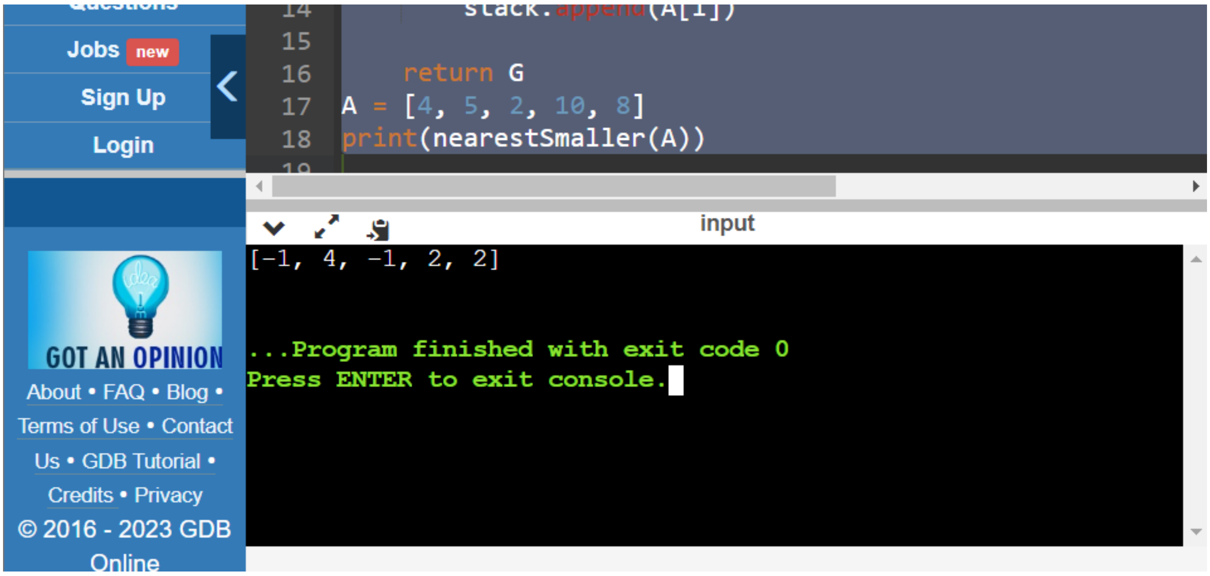
<u>Day-14 : Stack & Queue - 2</u>

Problem statement: Next Smaller Element

```
def nearestSmaller(A):
    stack = []
    G = []

    for i in range(len(A)):
        while stack and stack[-1] >= A[i]:
            stack.pop()

        if not stack:
            G.append(-1)
        else:
            G.append(stack[-1])

        stack.append(A[i])

    return G
A = [4, 5, 2, 10, 8]
print(nearestSmaller(A))
```



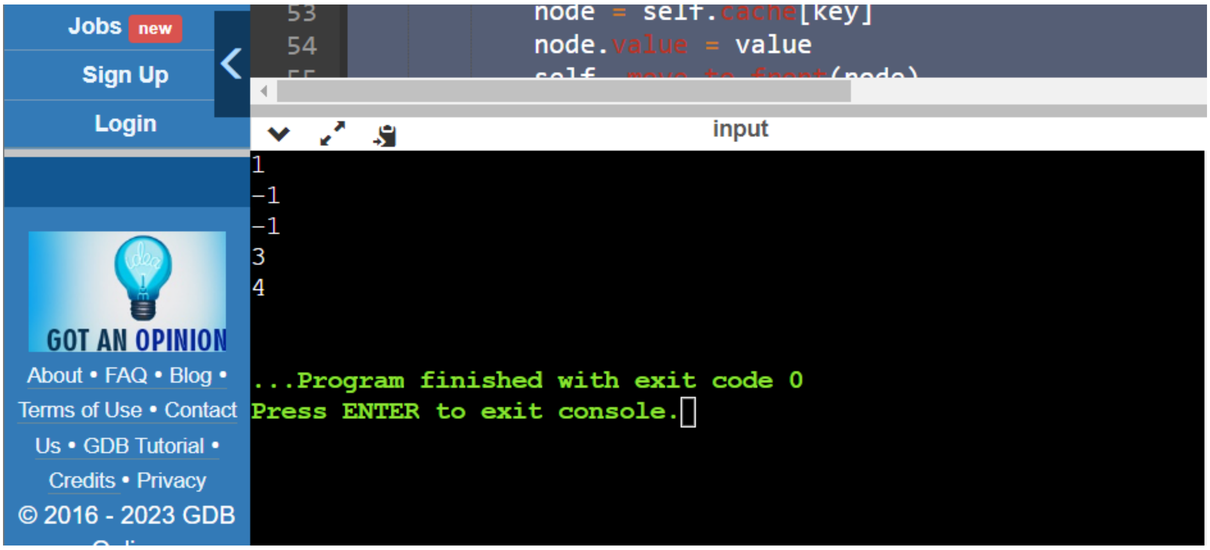
Problem Statement: Design a data structure that follows the constraints of **Least Recently Used (LRU) cache**

```
class LRUCache:

    class Node:
        def __init__(self, key=None, value=None):
            self.key = key
            self.value = value
            self.prev = None
            self.next = None

    def __init__(self, capacity):
        self.capacity = capacity
        self.cache = {}
        self.head = self.Node()
        self.tail = self.Node()
        self.head.next = self.tail
        self.tail.prev = self.head

    def _add_node(self, node):
```

```
    # Add a node to the front of the linked list
    node.prev = self.head
    node.next = self.head.next
    self.head.next.prev = node
    self.head.next = node

def _remove_node(self, node):
    # Remove a node from the linked list
    prev = node.prev
    next = node.next
    prev.next = next
    next.prev = prev

def _move_to_front(self, node):
    # Move a node to the front of the linked list
    self._remove_node(node)
    self._add_node(node)

def _pop_tail(self):
    # Remove and return the tail node from the linked list
    tail_node = self.tail.prev
    self._remove_node(tail_node)
    return tail_node

def get(self, key):
    if key in self.cache:
        node = self.cache[key]
        self._move_to_front(node)
        return node.value
    else:
        return -1
```

```
    def put(self, key, value):
        if key in self.cache:
            node = self.cache[key]
            node.value = value
            self._move_to_front(node)
        else:
            new_node = self.Node(key, value)
            self.cache[key] = new_node
            self._add_node(new_node)
            if len(self.cache) > self.capacity:
                tail_node = self._pop_tail()
                del self.cache[tail_node.key]
lru_cache = LRUCache(2)
lru_cache.put(1, 1)
lru_cache.put(2, 2)
print(lru_cache.get(1))
lru_cache.put(3, 3)
print(lru_cache.get(2))
lru_cache.put(4, 4)
print(lru_cache.get(1))
print(lru_cache.get(3))
```

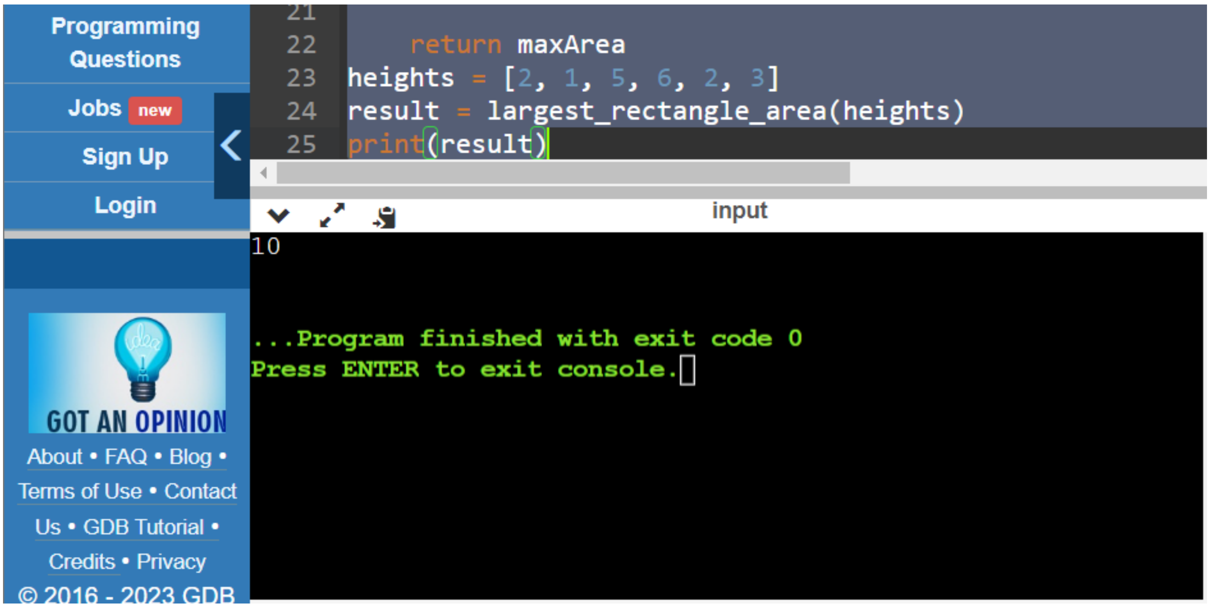
print(lru_cache.get(4))



Problem Statement: Given an array of integers heights representing the histogram's bar height where the width of each bar is 1 return the area of the largest rectangle in histogram.

```
def largest_rectangle_area(heights):
stack = []
maxArea = 0
i = 0

while i < len(heights):
    if not stack or heights[i] >= heights[stack[-1]]:
        stack.append(i)
        i += 1
    else:
        top = stack.pop()
        width = i if not stack else i - stack[-1] - 1
        area = heights[top] * width
        maxArea = max(maxArea, area)

while stack:
```

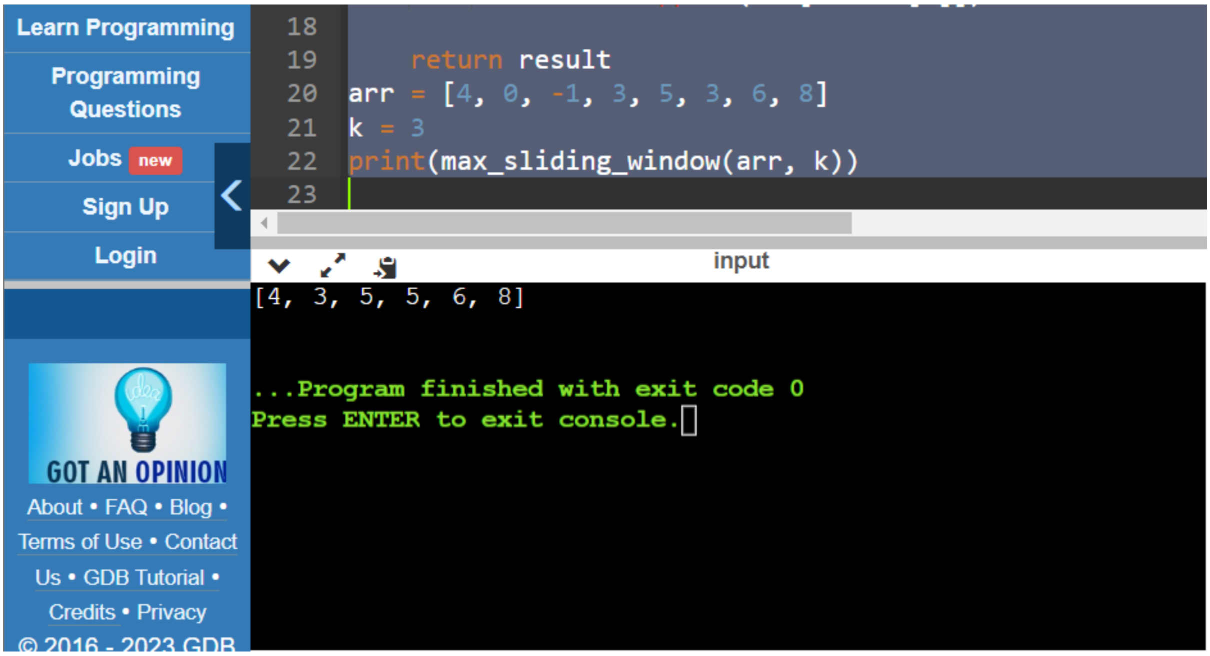
```
        top = stack.pop()
        width = i if not stack else len(heights) - stack[-1] - 1
        area = heights[top] * width
        maxArea = max(maxArea, area)

    return maxArea
heights = [2, 1, 5, 6, 2, 3]
result = largest_rectangle_area(heights)
print(result)
```



Problem Statement: Given an array of integers arr, there is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k numbers in the window. Each time the sliding window moves right by one position. Return *the **max sliding window***

```
from collections import deque


def max_sliding_window(arr, k):
    result = []
```

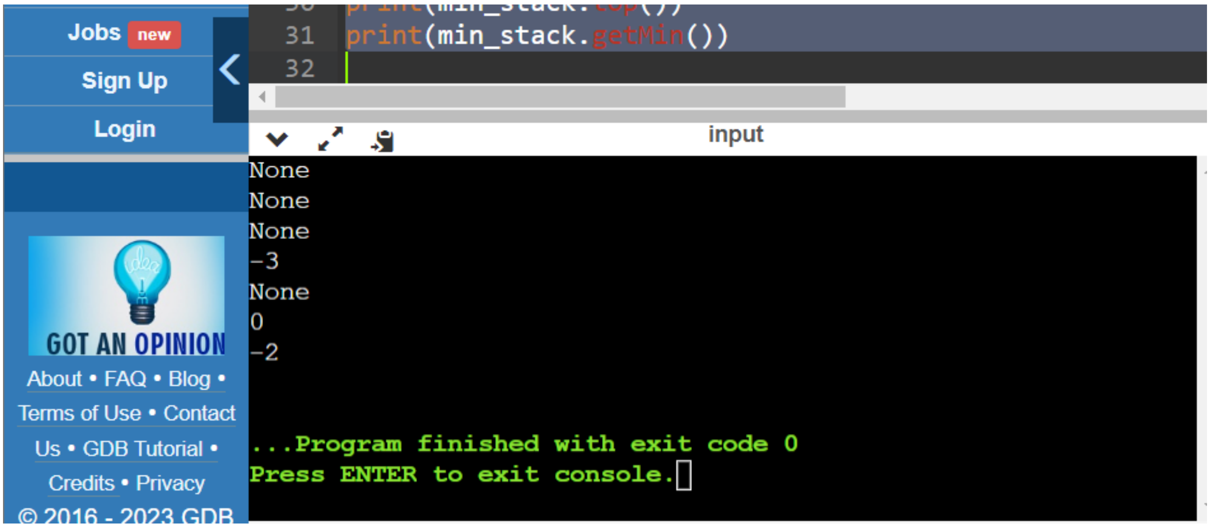
```
    window = deque()

    for i in range(len(arr)):
        if window and window[0] <= i - k:
            window.popleft()

        while window and arr[window[-1]] <= arr[i]:
            window.pop()

        window.append(i)

        if i >= k - 1:
            result.append(arr[window[0]])

    return result
arr = [4, 0, -1, 3, 5, 3, 6, 8]
k = 3
print(max_sliding_window(arr, k))
```



Problem Statement: Implement Min Stack | O(2N) and O(N) Space Complexity. Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

```
class MinStack:

    def __init__(self):

        self.stack = []

        self.min_stack = []



    def push(self, val):

        self.stack.append(val)

        if not self.min_stack or val <= self.min_stack[-1]:

            self.min_stack.append(val)



    def pop(self):
```

```
        if self.stack:
            val = self.stack.pop()
            if val == self.min_stack[-1]:
                self.min_stack.pop()

    def top(self):
        if self.stack:
            return self.stack[-1]

    def getMin(self):
        if self.min_stack:
            return self.min_stack[-1]
min_stack = MinStack()
print(min_stack.push(-2))
print(min_stack.push(0))
print(min_stack.push(-3))
print(min_stack.getMin())
print(min_stack.pop())
print(min_stack.top())
print(min_stack.getMin())
```



Problem Statement: You will be given an **m x n** grid, where each cell has the following values :

1. 2 – represents a rotten orange
2. 1 – represents a Fresh orange
3. 0 – represents an Empty Cell

Every minute, if a Fresh Orange is adjacent to a Rotten Orange in 4-direction (upward, downwards, right, and left) it becomes Rotten.

Return the minimum number of minutes required such that none of the cells has a Fresh Orange. If it’s not possible, return **-1**.

```
from collections import deque


def orangesRotting(grid):

    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

    queue = deque()

    fresh_oranges = 0

    minutes = 0
```

```
for i in range(len(grid)):
    for j in range(len(grid[0])):
        if grid[i][j] == 2:
            queue.append((i, j))
        elif grid[i][j] == 1:
            fresh_oranges += 1

while queue:
    size = len(queue)
    rotten_found = False

    for _ in range(size):
        x, y = queue.popleft()

        for dx, dy in directions:
            nx, ny = x + dx, y + dy

            if 0 <= nx < len(grid) and 0 <= ny < len(grid[0]) and grid[nx][ny] == 1:
                grid[nx][ny] = 2
                fresh_oranges -= 1
                queue.append((nx, ny))
                rotten_found = True
```

```
        if rotten_found:
            minutes += 1

    if fresh_oranges > 0:
        return -1
    else:
        return minutes

grid = [[2, 1, 1], [0, 1, 1], [1, 0, 1]]
print(orangesRotting(grid))
```