Stacks and Queues

Name: Sathvik Teja Moturi
Roll no. : MT2025717

Validate Stack Sequences

```
class Solution:
    def validateStackSequences(self, pushed, popped):
        stack = []
        j = 0
        for x in pushed:
            stack.append(x)
            while stack and j < len(popped) and stack[-1] == popped[j]:
                stack.pop()
                j += 1
        return not stack
```

Remove K Digits

```
class Solution:
    def removeKdigits(self, num: str, k: int) -> str:
        stack = []
        for digit in num:
            while stack and k > 0 and stack[-1] > digit:
                stack.pop()
                k -= 1
            stack.append(digit)
        while k > 0 and stack:
            stack.pop()
            k -= 1
        res = "".join(stack).lstrip("0")
        return res if res else "0"
```

Count of Smaller Numbers After Self

```
class Solution:
    def countSmaller(self, nums):
        n = len(nums)
        res = [0] * n
        stack = []  # will store (value, index)

        for i in range(n-1, -1, -1):
            count = 0
            # Pop all elements smaller than current
            while stack and stack[-1][0] < nums[i]:
                val, idx = stack.pop()
                count += 1
                res[idx] += 0  # popped elements don't change
            res[i] = count + (res[i] if res[i] else 0)

            # Push current back
            stack.append((nums[i], i))

        return res
```

132 pattern

```
class Solution:
    def find132pattern(self, nums):
        stack = []
        third = float('-inf')
        for num in reversed(nums):
            if num < third:
                return True
            while stack and num > stack[-1]:
                third = stack.pop()
            stack.append(num)
        return False
```

Online Stock Span

```
class StockSpanner:
    def __init__(self):
        self.stack = []  # (price, span)

    def next(self, price: int) -> int:
        span = 1
        while self.stack and self.stack[-1][0] <= price:
            span += self.stack.pop()[1]
        self.stack.append((price, span))
        return span
```

Next Greater Element II

```
class Solution:
    def nextGreaterElements(self, nums):
        n = len(nums)
        res = [-1] * n
        stack = []
        for i in range(2*n):
            while stack and nums[stack[-1]] < nums[i % n]:
                res[stack.pop()] = nums[i % n]
            if i < n:
                stack.append(i)
        return res
```

Maximal Rectangle

```
class Solution:
    def maximalRectangle(self, matrix):
        if not matrix: return 0
        n = len(matrix[0])
        heights = [0]*n
        res = 0
        for row in matrix:
            for i in range(n):
                heights[i] = heights[i]+1 if row[i]=='1' else 0
            res = max(res, self.largestRectangleArea(heights))
        return res

    def largestRectangleArea(self, heights):
        stack = []
        res = 0
        heights.append(0)
        for i, h in enumerate(heights):
            while stack and heights[stack[-1]] > h:
                height = heights[stack.pop()]
                width = i if not stack else i - stack[-1] - 1
                res = max(res, height*width)
            stack.append(i)
        return res
```

Largest Rectangle in Histogram

```
class Solution:
    def largestRectangleArea(self, heights):
        stack = []
        res = 0
        heights.append(0)
        for i, h in enumerate(heights):
            while stack and heights[stack[-1]] > h:
                height = heights[stack.pop()]
                width = i if not stack else i - stack[-1] - 1
                res = max(res, height*width)
            stack.append(i)
        return res
```

Trapping Rain Water

```
class Solution:
    def trap(self, height):
        stack = []
        water = 0
        for i, h in enumerate(height):
            while stack and h > height[stack[-1]]:
                top = stack.pop()
                if not stack: break
                dist = i - stack[-1] - 1
                bounded = min(h, height[stack[-1]]) - height[top]
                water += dist * bounded
            stack.append(i)
        return water
```

Beautiful Towers I

```
class Solution:
    def maximumSumOfHeights(self, maxHeights):
        n = len(maxHeights)
        res = 0
        for peak in range(n):
            s = maxHeights[peak]
            h = maxHeights[peak]
            for i in range(peak-1,-1,-1):
                h = min(h, maxHeights[i])
                s += h
            h = maxHeights[peak]
            for i in range(peak+1,n):
                h = min(h, maxHeights[i])
                s += h
            res = max(res,s)
        return res
```

Beautiful Towers II

```
class Solution:
    def maximumSumOfHeights(self, maxHeights):
        n = len(maxHeights)
        left, stack = [0]*n, []
        for i in range(n):
            h = maxHeights[i]
            while stack and maxHeights[stack[-1]] > h:
                stack.pop()
            if stack:
                left[i] = left[stack[-1]] + (i-stack[-1])*h
            else:
                left[i] = (i+1)*h
            stack.append(i)

        right, stack = [0]*n, []
        for i in range(n-1,-1,-1):
            h = maxHeights[i]
            while stack and maxHeights[stack[-1]] > h:
                stack.pop()
            if stack:
                right[i] = right[stack[-1]] + (stack[-1]-i)*h
            else:
                right[i] = (n-i)*h
            stack.append(i)

        return max(left[i]+right[i]-maxHeights[i] for i in range(n))
```

Design Circular Deque

```
class MyCircularDeque:
    def __init__(self, k: int):
        self.q = [0]*k
        self.k = k
        self.size = 0
        self.front = 0
        self.rear = -1

    def insertFront(self, value: int) -> bool:
        if self.isFull(): return False
        self.front = (self.front-1)%self.k
        self.q[self.front] = value
        self.size += 1
        if self.size == 1: self.rear = self.front
        return True

    def insertLast(self, value: int) -> bool:
        if self.isFull(): return False
        self.rear = (self.rear+1)%self.k
        self.q[self.rear] = value
        self.size += 1
        if self.size == 1: self.front = self.rear
        return True

    def deleteFront(self) -> bool:
        if self.isEmpty(): return False
        self.front = (self.front+1)%self.k
        self.size -= 1
        return True

    def deleteLast(self) -> bool:
        if self.isEmpty(): return False
        self.rear = (self.rear-1)%self.k
        self.size -= 1
        return True

    def getFront(self) -> int:
        return -1 if self.isEmpty() else self.q[self.front]

    def getRear(self) -> int:
        return -1 if self.isEmpty() else self.q[self.rear]

    def isEmpty(self) -> bool:
        return self.size == 0

    def isFull(self) -> bool:
        return self.size == self.k
```

Sliding Window

```
from collections import deque

class Solution:
    def maxSlidingWindow(self, nums, k):
        dq = deque()
        res = []
        for i, num in enumerate(nums):
            while dq and dq[0] <= i-k:
                dq.popleft()
            while dq and nums[dq[-1]] <= num:
                dq.pop()
            dq.append(i)
            if i >= k-1:
                res.append(nums[dq[0]])
        return res
```

Count Subarrays with Fixed

```
class Solution:
    def countSubarrays(self, nums, minK, maxK):
        res = 0
        left = min_pos = max_pos = -1
        for i, num in enumerate(nums):
            if not minK <= num <= maxK:
                left = i
            if num == minK:
                min_pos = i
            if num == maxK:
                max_pos = i
            res += max(0, min(min_pos, max_pos) - left)
        return res
```

MK averageBounds

```
class MKAverage:
    def __init__(self, m: int, k: int):
        self.m = m
        self.k = k
        self.stream = []   # keep last m elements

    def addElement(self, num: int) -> None:
        self.stream.append(num)
        if len(self.stream) > self.m:
            self.stream.pop(0)  # remove oldest element

    def calculateMKAverage(self) -> int:
        if len(self.stream) < self.m:
            return -1
        arr = sorted(self.stream)  # sort copy
        middle = arr[self.k : self.m - self.k]  # remove k smallest and k largest
        return sum(middle) // len(middle)
```