**ADVANCE CODING – 4**

**V.Venkatesh**

**VU21CSEN0300024**

1. **Maximum Sum Circular Subarray**

**Code:**

class Solution:

    def maxSubarraySumCircular(self, nums: list[int]) -> int:

        def kadane(arr):

            max\_sum = current\_sum = arr[0]

            for num in arr[1:]:

                current\_sum = max(num, current\_sum + num)

                max\_sum = max(max\_sum, current\_sum)

            return max\_sum

        total\_sum = sum(nums)

        max\_sum = kadane(nums)

        # Invert nums for finding the minimum subarray sum

        min\_sum = kadane([-num for num in nums])

        # Handle the case where all numbers are negative

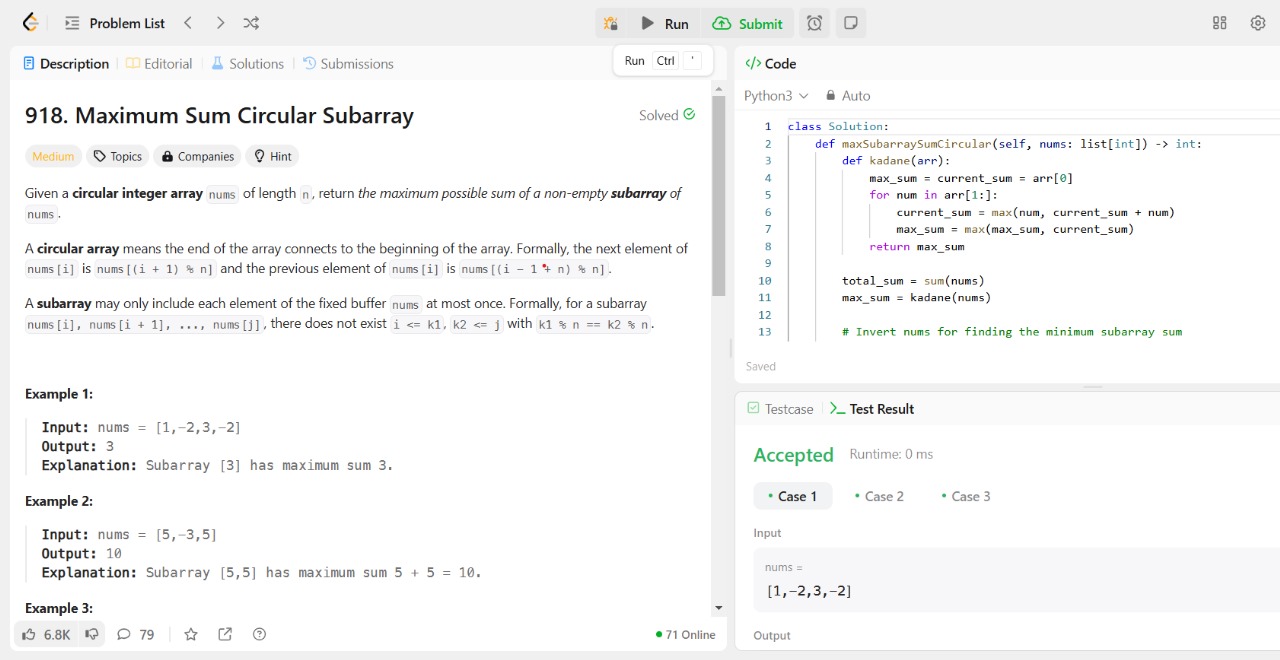
        if max\_sum < 0:

            return max\_sum

        # Maximum of non-circular and circular cases

        return max(max\_sum, total\_sum + min\_sum)

**Output:**



1. **Stamping The Sequence**

**Code:**

class Solution:

    def movesToStamp(self, stamp: str, target: str) -> list[int]:

        m, n = len(stamp), len(target)

        target\_list = list(target)

        s = ['?'] \* n

        result = []

        made\_change = True

        def can\_stamp(start):

            # Check if the stamp can be placed at position 'start'

            can\_place = False

            for i in range(m):

                if target\_list[start + i] != '?' and target\_list[start + i] != stamp[i]:

                    return False

                if target\_list[start + i] != '?':

                    can\_place = True

            return can\_place

        def do\_stamp(start):

            # Place the stamp at position 'start' and replace with '?'

            for i in range(m):

                target\_list[start + i] = '?'

        # Main simulation loop

        while made\_change:

            made\_change = False

            for i in range(n - m + 1):  # Try placing stamp at each valid position

                if can\_stamp(i):

                    do\_stamp(i)

                    result.append(i)

                    made\_change = True

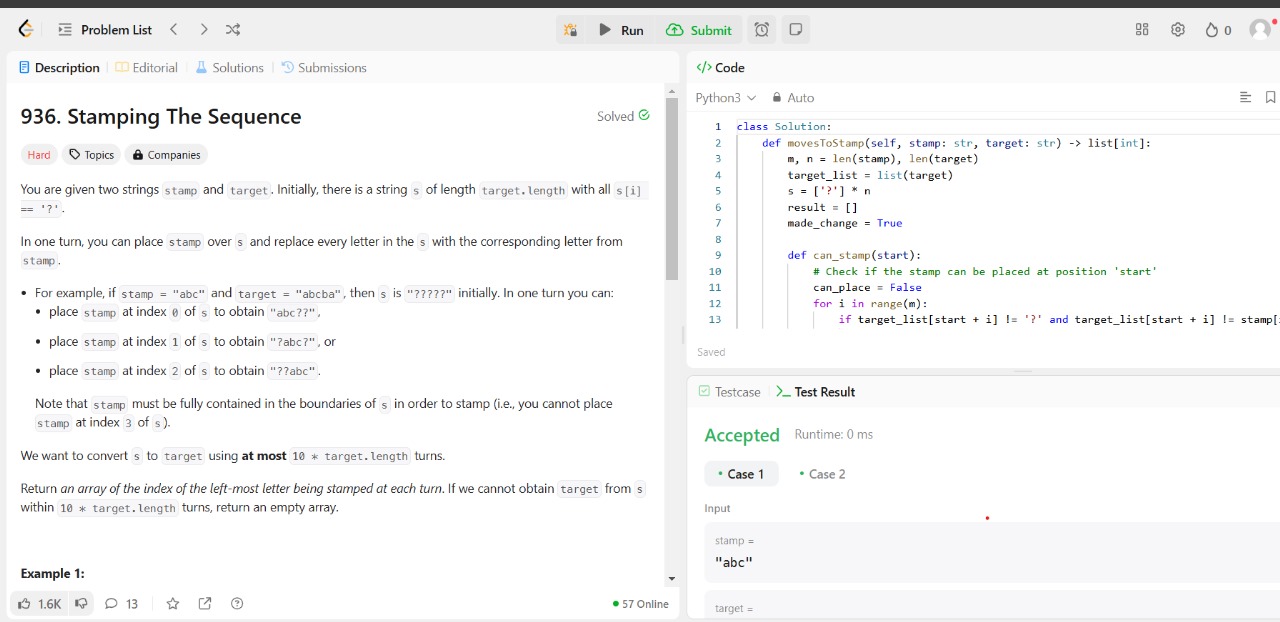
        # Check if the transformation is complete

        if any(c != '?' for c in target\_list):

            return []

        return result[::-1]

**Output:**

****

1. **Design Browser History**

class BrowserHistory:

def \_\_init\_\_(self, homepage: str):

self.history = [homepage]

self.curr = 0

self.bound = 0

def visit(self, url: str) -> None:

self.curr += 1

if self.curr == len(self.history):

self.history.append(url)

else:

self.history[self.curr] = url

self.bound = self.curr

def back(self, steps: int) -> str:

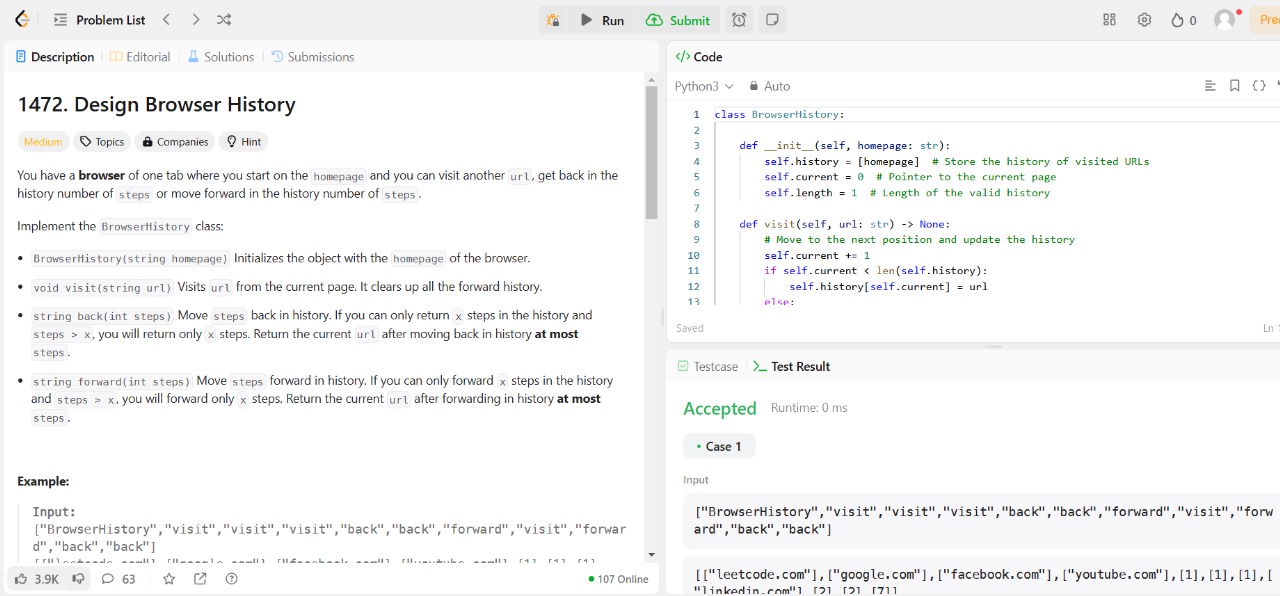
self.curr = max(self.curr - steps, 0)

return self.history[self.curr]

def forward(self, steps: int) -> str:

self.curr = min(self.curr + steps, self.bound)

return self.history[self.curr]



**4.LRU Cache**

class LRUCache:

class Node:

def \_\_init\_\_(self, key, val):

self.key = key

self.val = val

self.prev = None

self.next = None

def \_\_init\_\_(self, capacity: int):

self.cap = capacity

self.head = self.Node(-1, -1)

self.tail = self.Node(-1, -1)

self.head.next = self.tail

self.tail.prev = self.head

self.m = {}

def addNode(self, newnode):

temp = self.head.next

newnode.next = temp

newnode.prev = self.head

self.head.next = newnode

temp.prev = newnode

def deleteNode(self, delnode):

prevv = delnode.prev

nextt = delnode.next

prevv.next = nextt

nextt.prev = prevv

def get(self, key: int) -> int:

if key in self.m:

resNode = self.m[key]

ans = resNode.val

del self.m[key]

self.deleteNode(resNode)

self.addNode(resNode)

self.m[key] = self.head.next

return ans

return -1

def put(self, key: int, value: int) -> None:

if key in self.m:

curr = self.m[key]

del self.m[key]

self.deleteNode(curr)

if len(self.m) == self.cap:

del self.m[self.tail.prev.key]

self.deleteNode(self.tail.prev)

self.addNode(self.Node(key, value))

self.m[key] = self.head.next

