**1763.Longest Nice Substring**

**from collections import defaultdict**

**class Solution:**

**def longestNiceSubstring(self, s):**

**len\_s = len(s)**

**if len\_s <= 1:**

**return ''**

**char\_to\_freq\_map = defaultdict(int)**

**for c in s:**

**char\_to\_freq\_map[c] += 1**

**is\_broken = False**

**i = 0**

**while (i < len(s)):**

**if s[i].islower() and s[i].upper() in char\_to\_freq\_map.keys():**

**pass**

**elif s[i].isupper() and s[i].lower() in char\_to\_freq\_map.keys():**

**pass**

**else:**

**is\_broken = True**

**break**

**i += 1**

**if not is\_broken:**

**return s**

**longest\_nice\_substr\_1 = self.longestNiceSubstring(s[:i])**

**longest\_nice\_substr\_2 = self.longestNiceSubstring(s[i+1:])**

**if len(longest\_nice\_substr\_1)>=len(longest\_nice\_substr\_2):**

**return longest\_nice\_substr\_1**

**else:**

**return longest\_nice\_substr\_2**

****

**190.Reverse Bits**

**class Solution:**

**def reverseBits(self, n):**

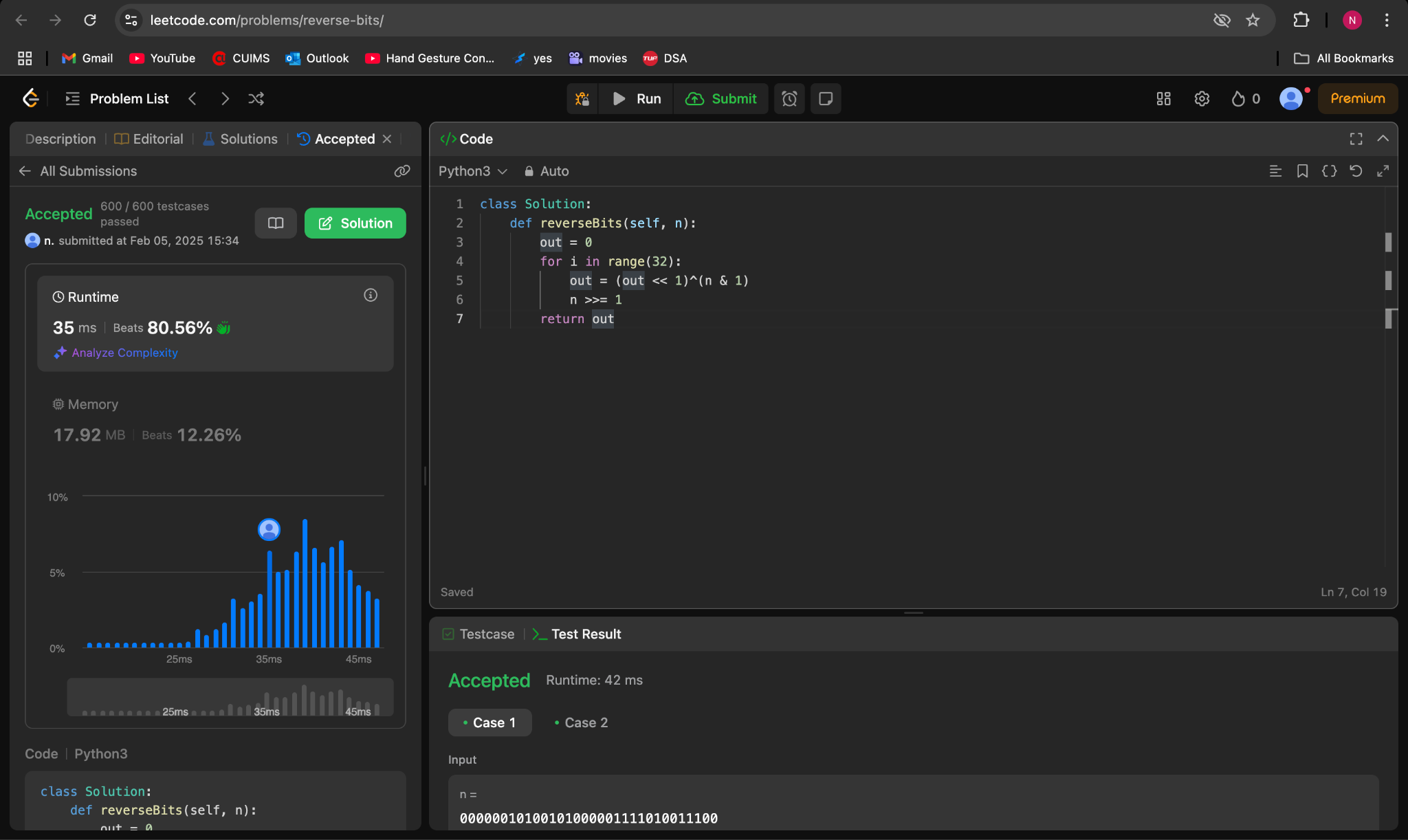
**out = 0**

**for i in range(32):**

**out = (out << 1)^(n & 1)**

**n >>= 1**

**return out**

****

**191.Number of 1 Bits**

**class Solution:**

**def hammingWeight(self, n: int) -> int:**

**bin\_n = bin(n)[2:]**

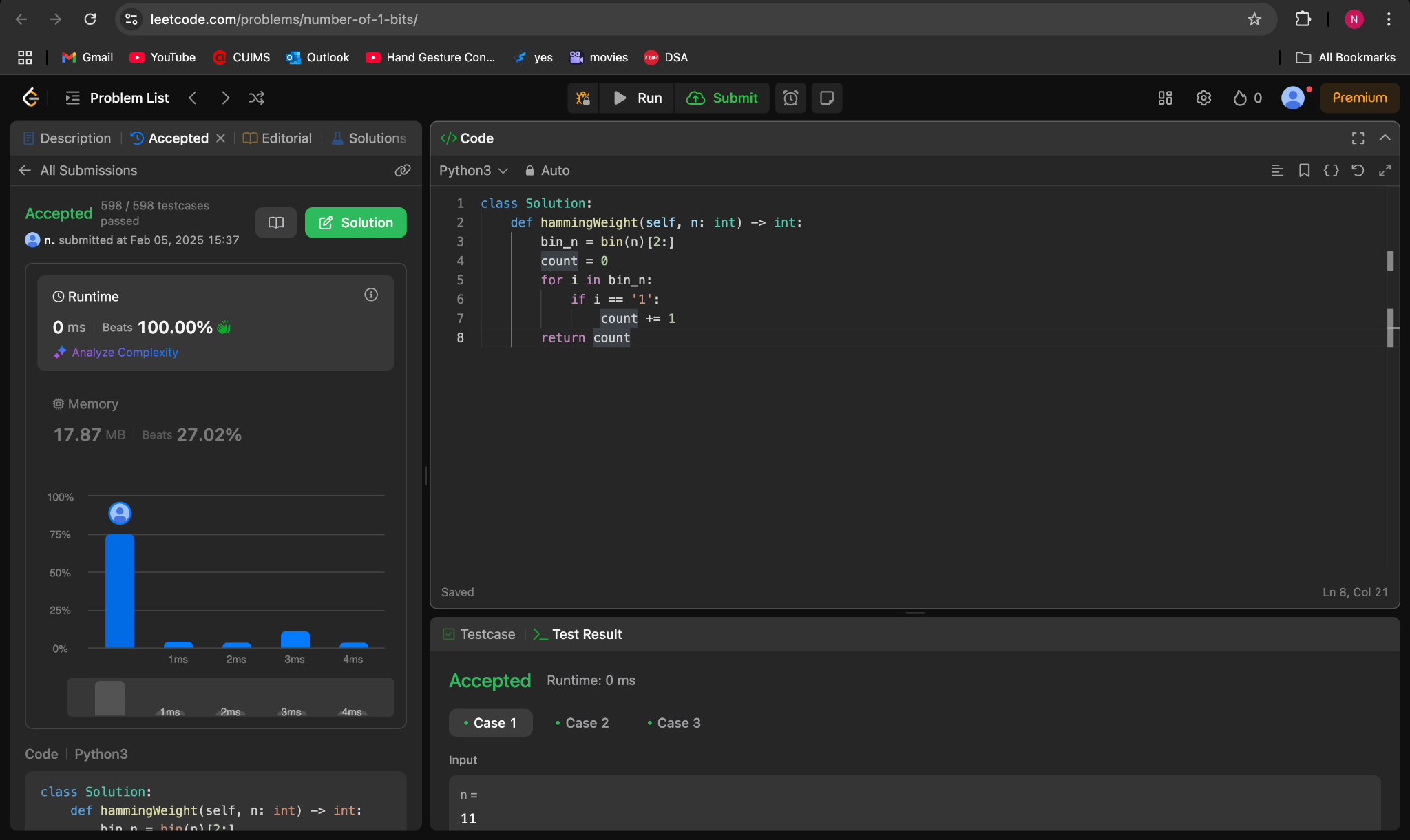
**count = 0**

**for i in bin\_n:**

**if i == '1':**

**count += 1**

**return count**

****

**53.Maximum Subarray**

**class Solution:**

**def maxSubArray(self, nums: List[int]) -> int:**

**res = nums[0]**

**total = 0**

**for n in nums:**

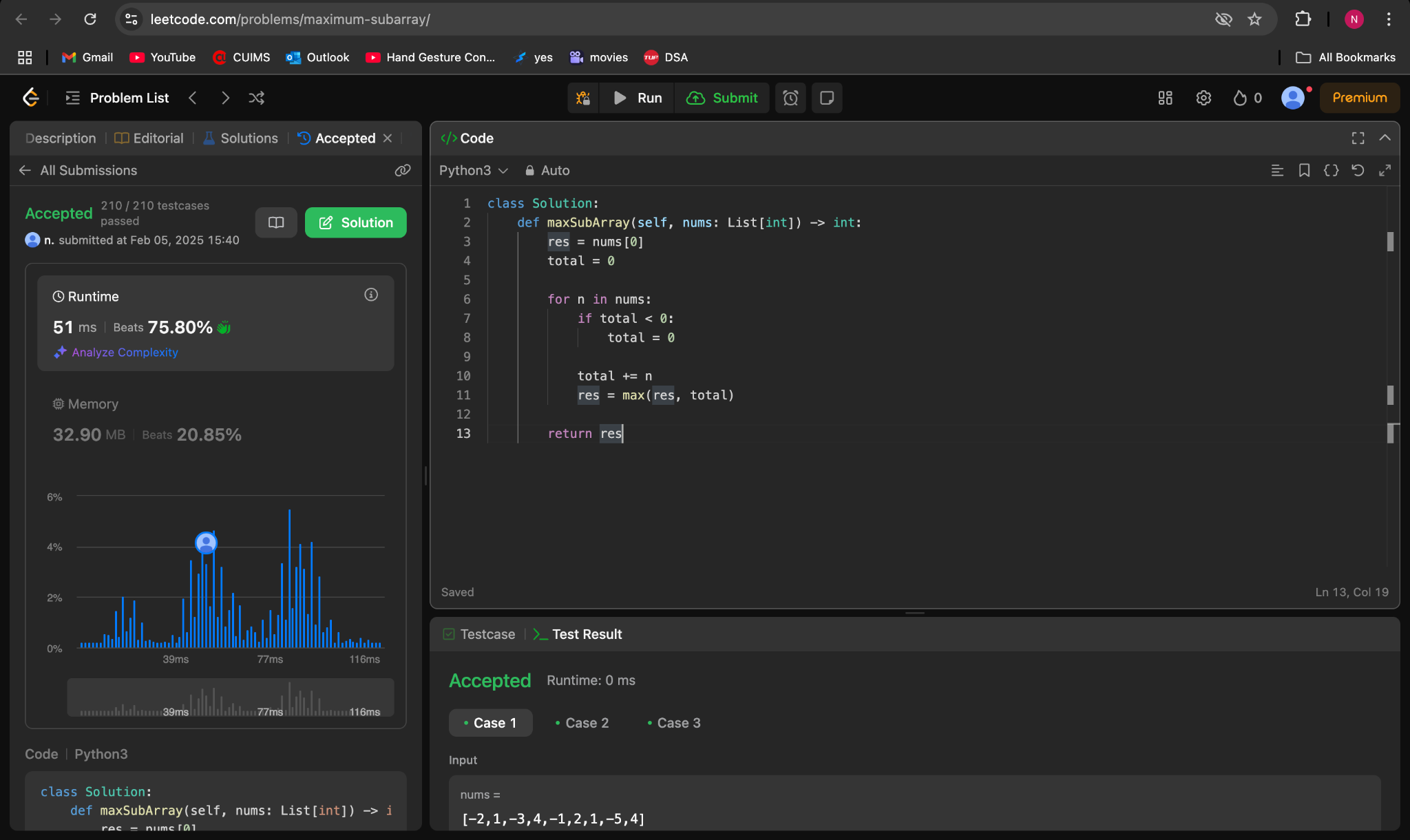
**if total < 0:**

**total = 0**

**total += n**

**res = max(res, total)**

**return res**

****

**240.Search a 2D Matrix II**

**class Solution:**

**def searchMatrix(self, mat: List[List[int]], target: int) -> bool:**

**m=len(mat)**

**n=len(mat[0])**

**for i in range(m):**

**if mat[i][0]<=target and mat[i][-1]>=target:**

**lo=0**

**hi=n**

**while (lo<hi):**

**mid=(lo+hi)//2**

**if mat[i][mid]==target:**

**return True**

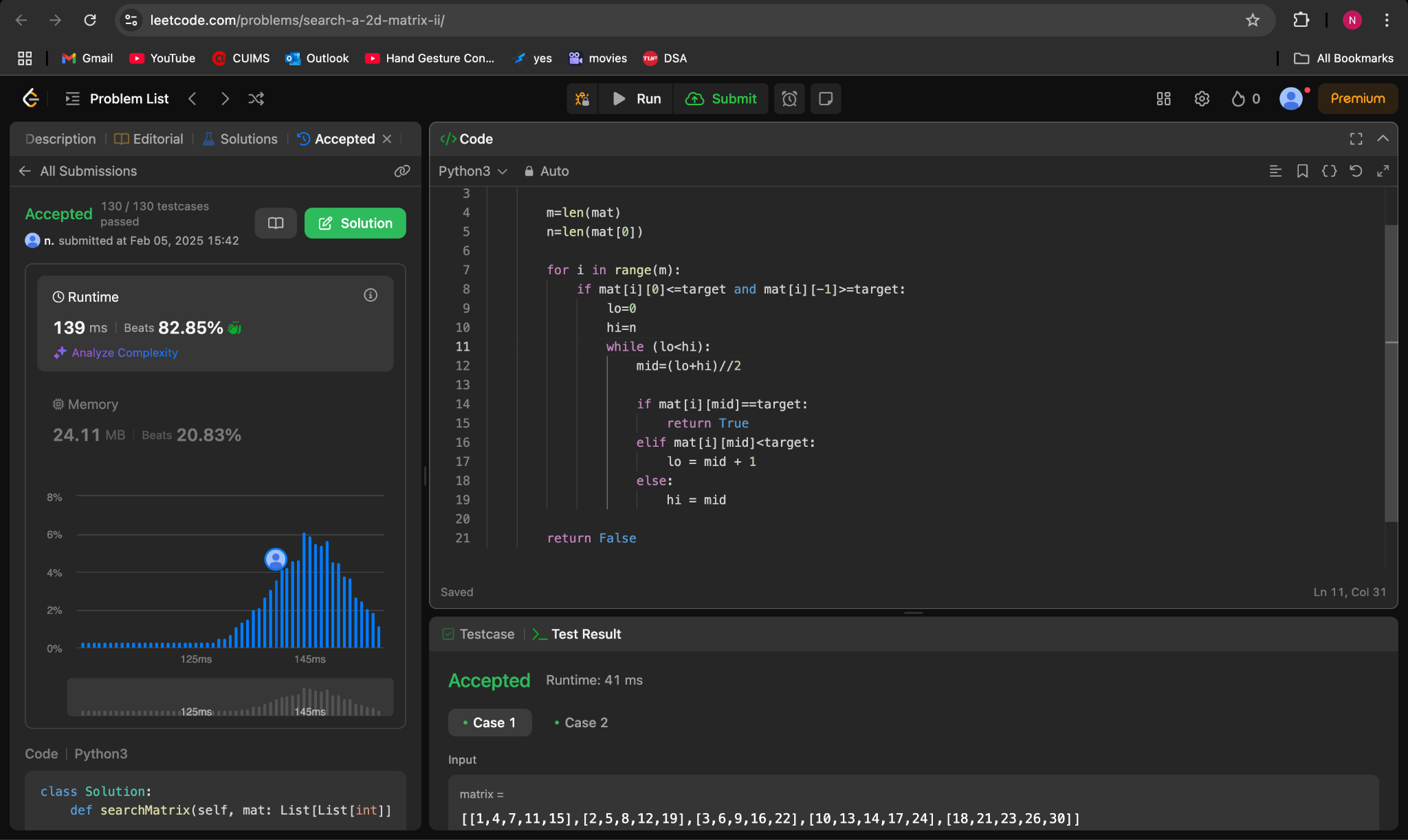
**elif mat[i][mid]<target:**

**lo = mid + 1**

**else:**

**hi = mid**

**return False**

****

**372.Super Pow**

**class Solution:**

**def superPow(self, a: int, b: List[int]) -> int:**

**mod = 1337**

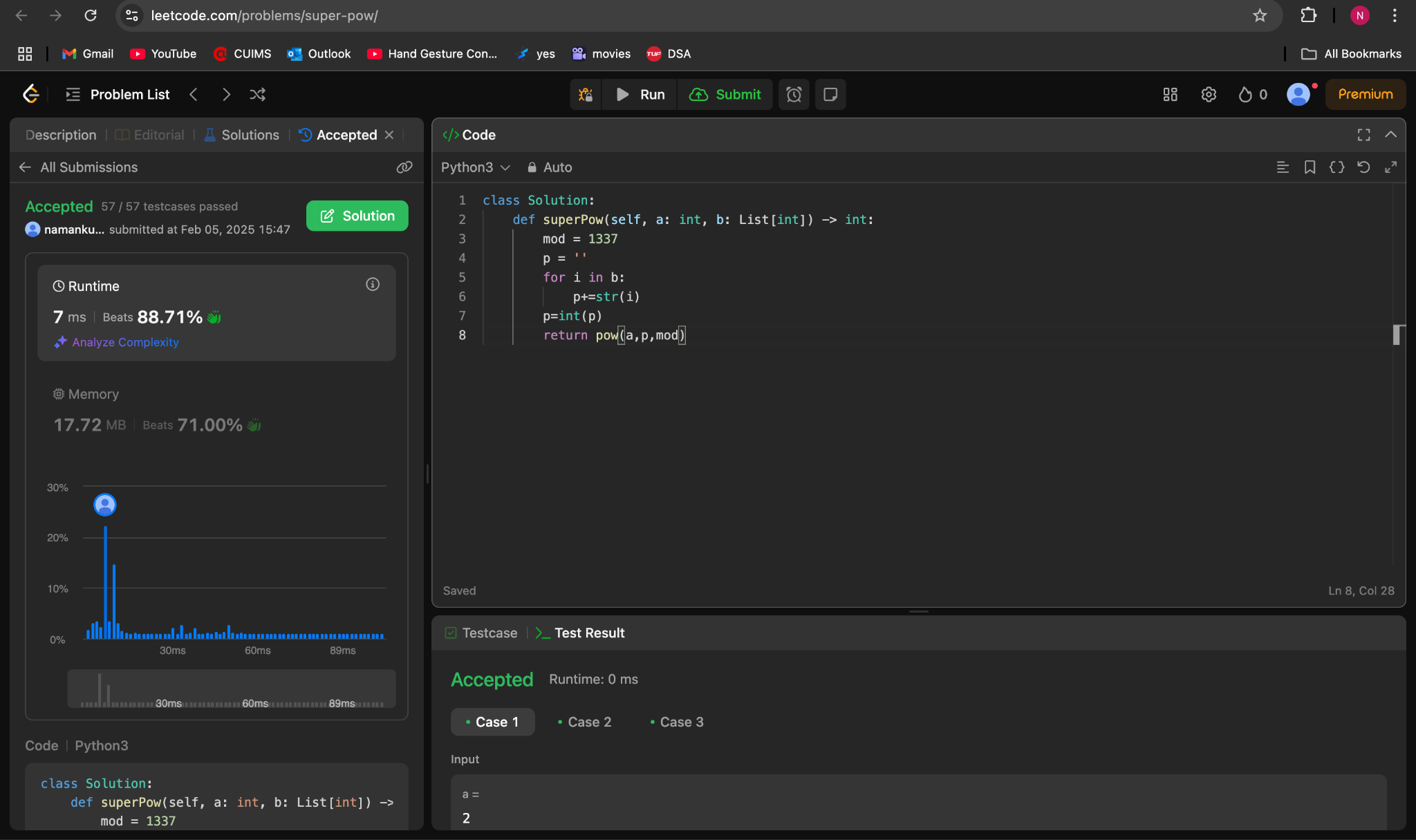
**p = ''**

**for i in b:**

**p+=str(i)**

**p=int(p)**

**return pow(a,p,mod)**

****

**932.Beautiful Array**

**class Solution:**

**def beautifulArray(self, N):**

**@lru\_cache(None)**

**def dfs(N):**

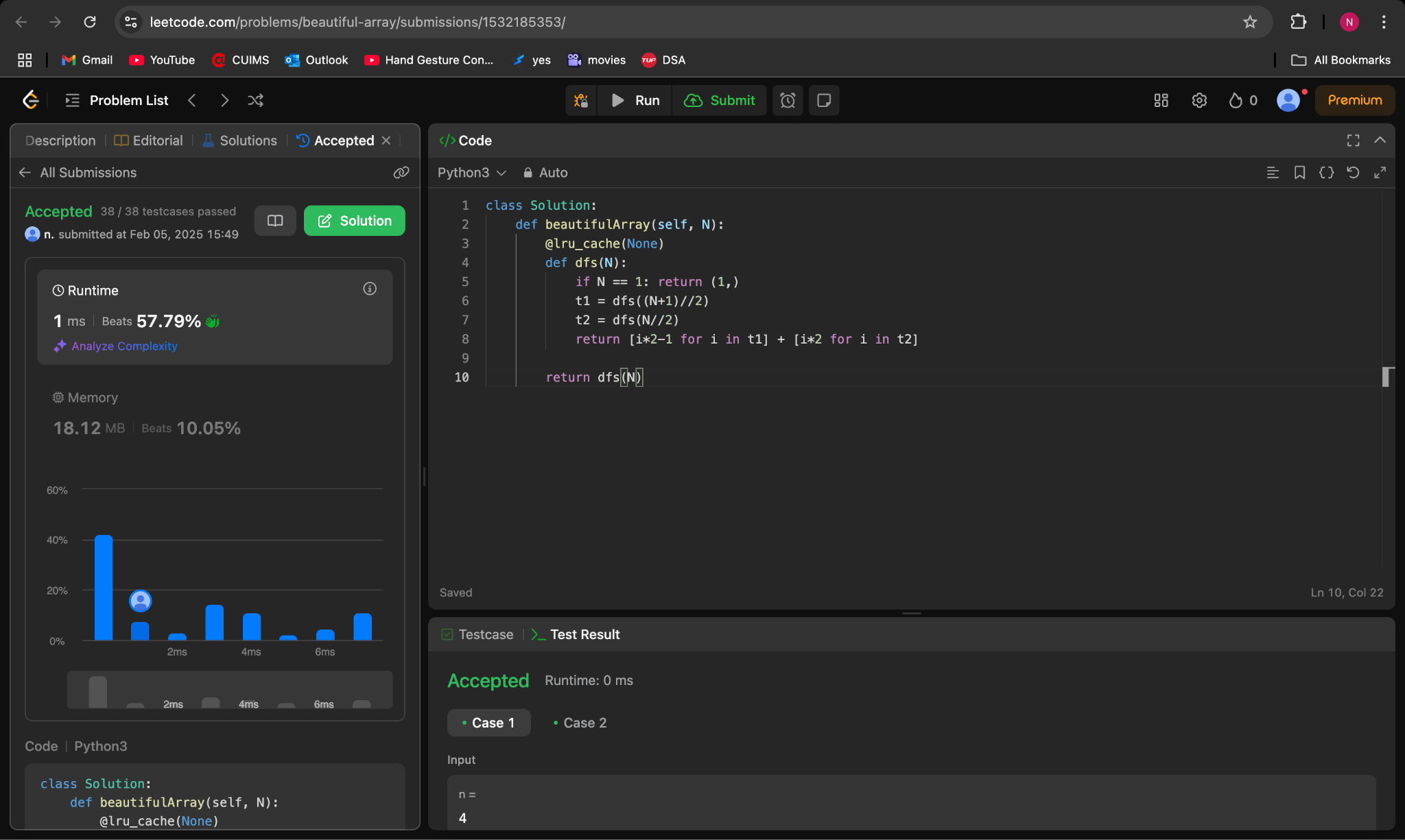
**if N == 1: return (1,)**

**t1 = dfs((N+1)//2)**

**t2 = dfs(N//2)**

**return [i\*2-1 for i in t1] + [i\*2 for i in t2]**

**return dfs(N)**

****

**218.The Skyline Problem**

**class Solution(object):**

**def getSkyline(self, buildings):**

**"""**

**:type buildings: List[List[int]]**

**:rtype: List[List[int]]**

**"""**

**def addsky(pos, hei):**

**if sky[-1][1] != hei:**

**sky.append([pos, hei])**

**sky = [[-1,0]]**

**position = set([b[0] for b in buildings] + [b[1] for b in buildings])**

**live = []**

**i = 0**

**for t in sorted(position):**

**while i < len(buildings) and buildings[i][0] <= t:**

**heappush(live, (-buildings[i][2], buildings[i][1]))**

**i += 1**

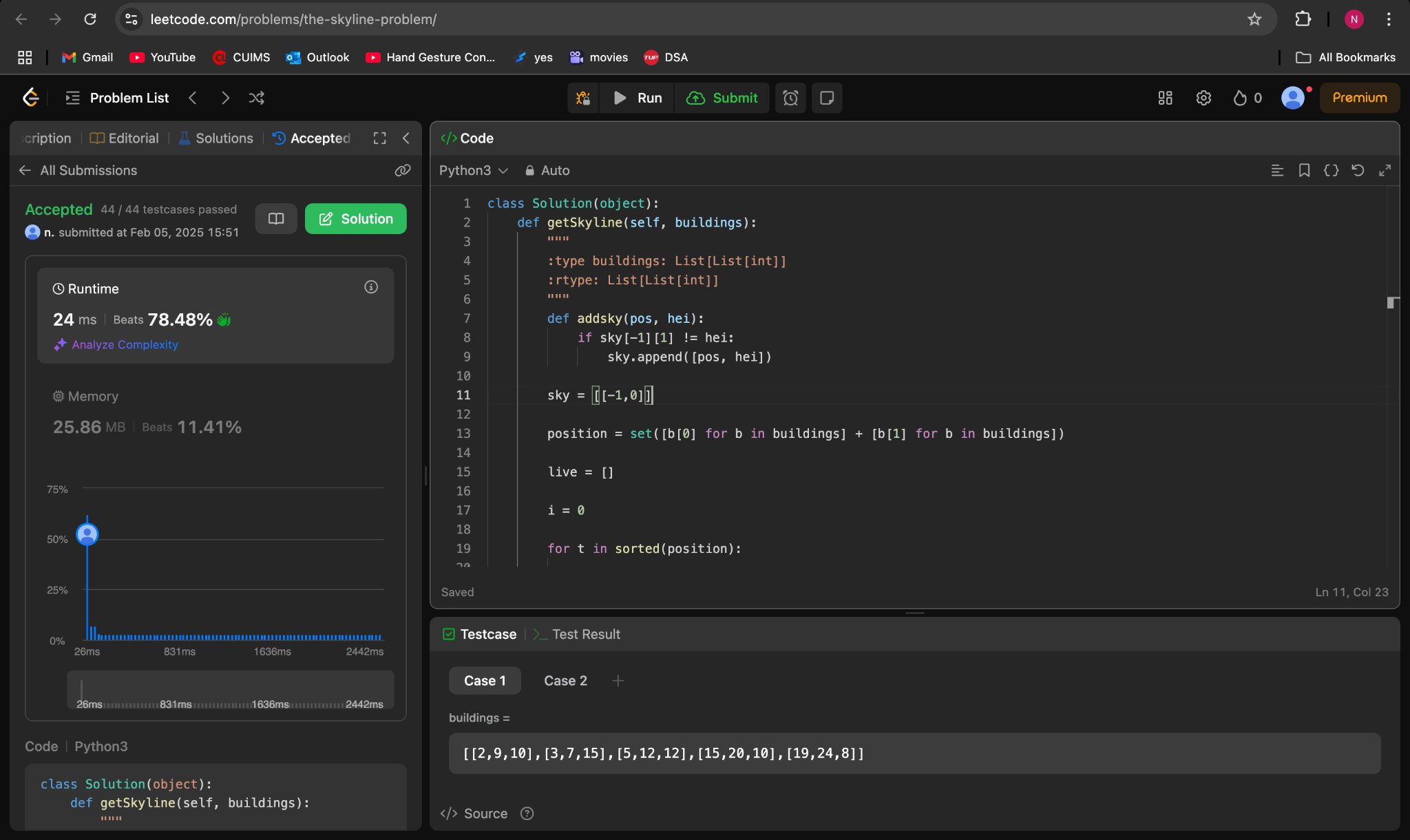
**while live and live[0][1] <= t:**

**heappop(live)**

**h = -live[0][0] if live else 0**

**addsky(t, h)**

**return sky[1:]**

****

**493.Reverse Pairs**

**from sortedcontainers import SortedList**

**class Solution:**

**def reversePairs(self, nums: List[int]) -> int:**

**n = len(nums)**

**l = SortedList()**

**res = 0**

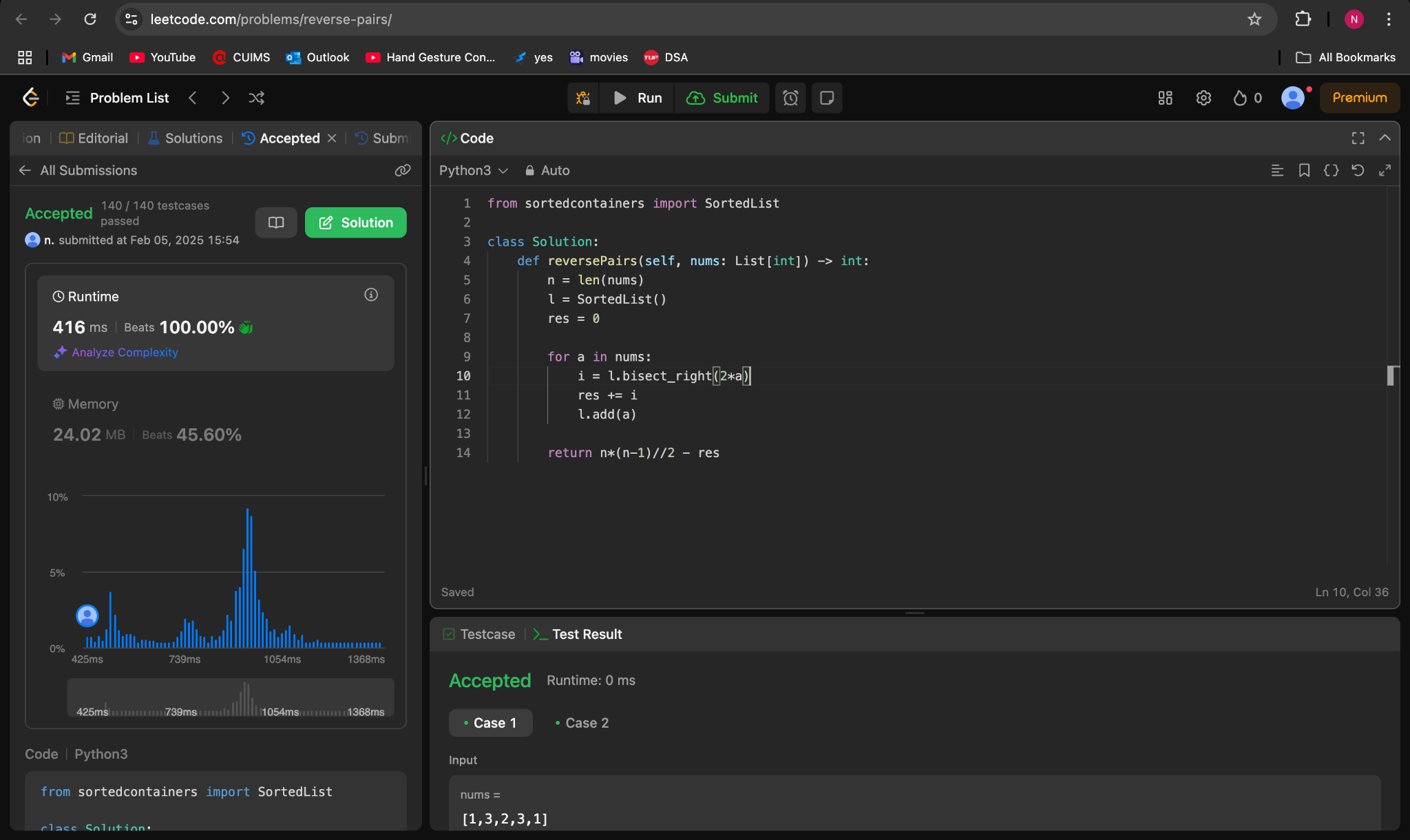
**for a in nums:**

**i = l.bisect\_right(2\*a)**

**res += i**

**l.add(a)**

**return n\*(n-1)//2 - res**

****

**2407.Longest Increasing Subsequence II**

**class SEG:**

**def \_\_init\_\_(self, n):**

**self.n = n**

**self.tree = [0] \* 2 \* self.n**

**def query(self, l, r):**

**l += self.n**

**r += self.n**

**ans = 0**

**while l < r:**

**if l & 1:**

**ans = max(ans, self.tree[l])**

**l += 1**

**if r & 1:**

**r -= 1**

**ans = max(ans, self.tree[r])**

**l >>= 1**

**r >>= 1**

**return ans**

**def update(self, i, val):**

**i += self.n**

**self.tree[i] = val**

**while i > 1:**

**i >>= 1**

**self.tree[i] = max(self.tree[i \* 2], self.tree[i \* 2 + 1])**

**class Solution:**

**def lengthOfLIS(self, A: List[int], k: int) -> int:**

**n, ans = max(A), 1**

**seg = SEG(n)**

**for a in A:**

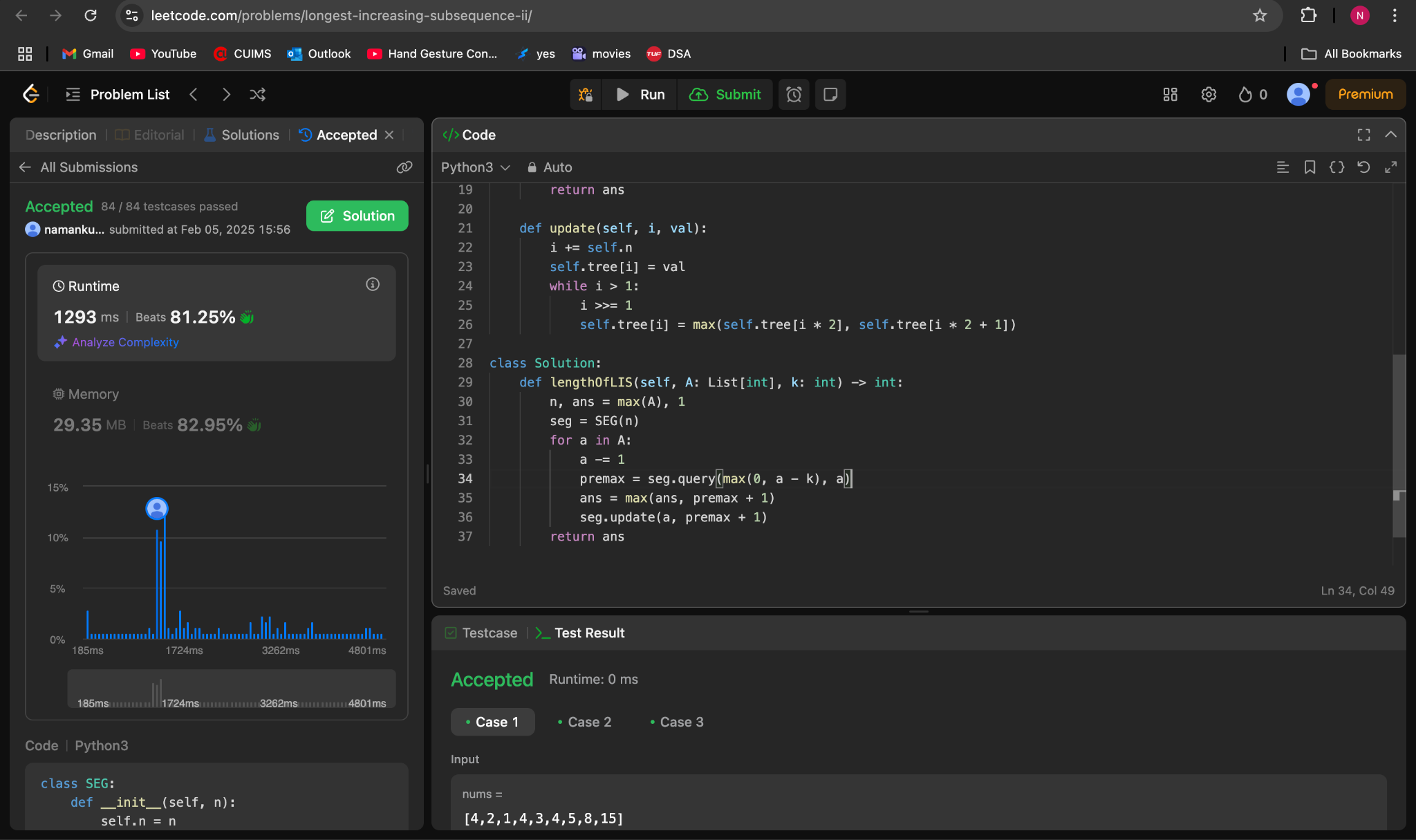
**a -= 1**

**premax = seg.query(max(0, a - k), a)**

**ans = max(ans, premax + 1)**

**seg.update(a, premax + 1)**

**return ans**

****

**88.Merge Sorted Array**

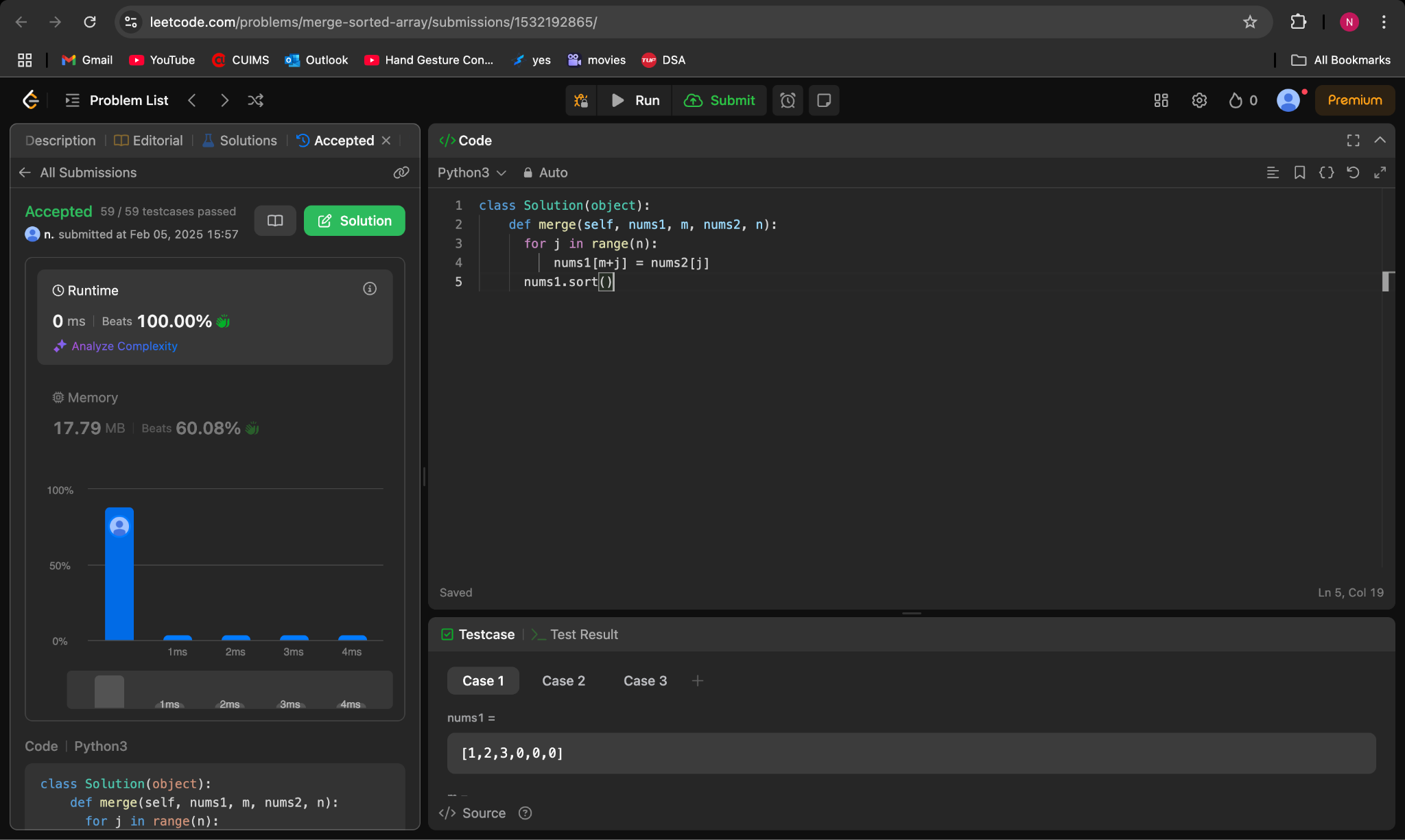
**class Solution(object):**

**def merge(self, nums1, m, nums2, n):**

**for j in range(n):**

**nums1[m+j] = nums2[j]**

**nums1.sort()**

****

**278.First Bad Version**

**class Solution:**

**def firstBadVersion(self, n: int) -> int:**

**left ,right = 0, n -1**

**calls = 0**

**while left <= right:**

**mid = (left + right) // 2**

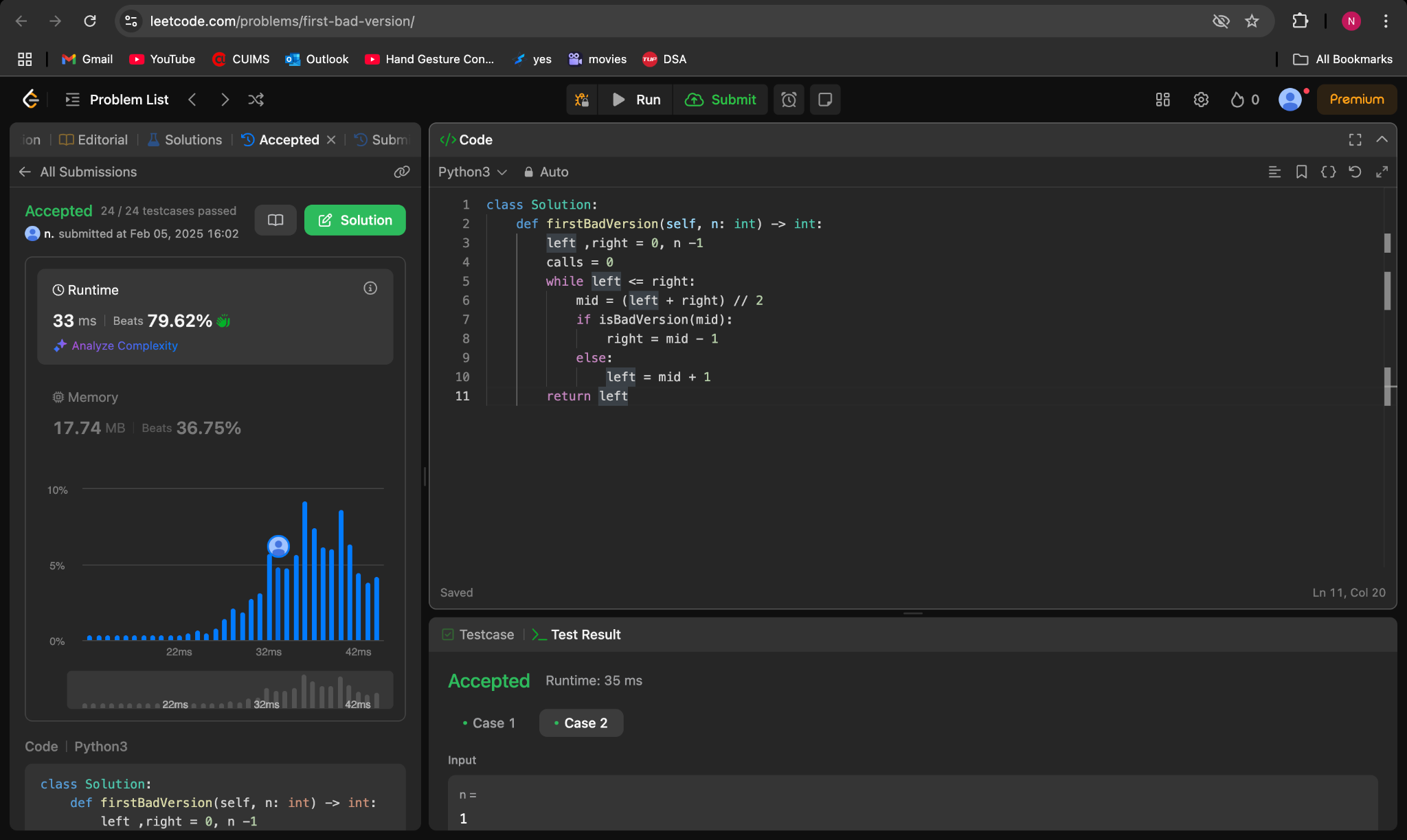
**if isBadVersion(mid):**

**right = mid - 1**

**else:**

**left = mid + 1**

**return left**

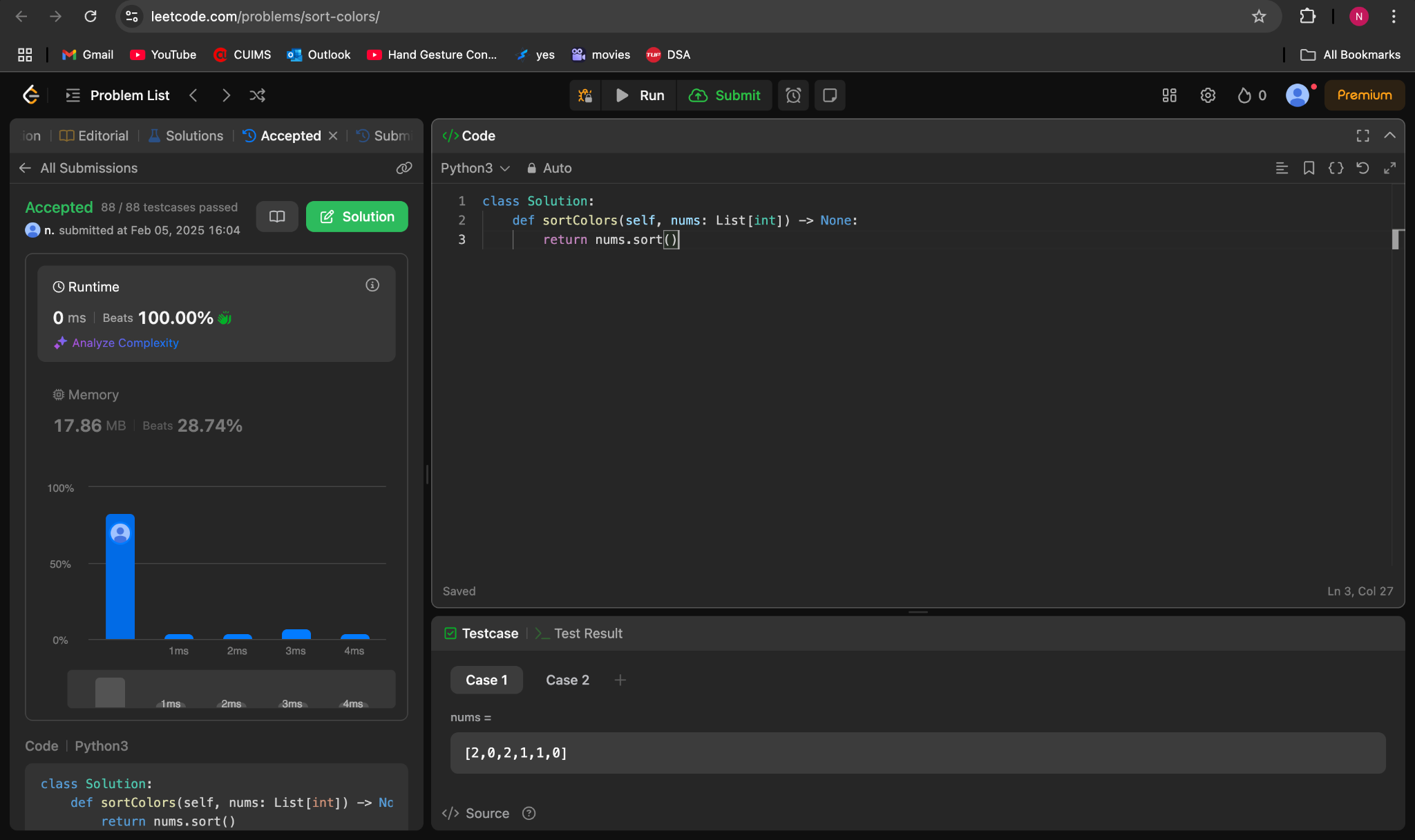
****

**75.Sort Colors**

**class Solution:**

**def sortColors(self, nums: List[int]) -> None:**

**return nums.sort()**

****

**347.Top K Frequent Elements**

**class Solution:**

**def topKFrequent(self, nums: List[int], k: int) -> List[int]:**

**freq = {}**

**for num in nums:**

**if num in freq:**

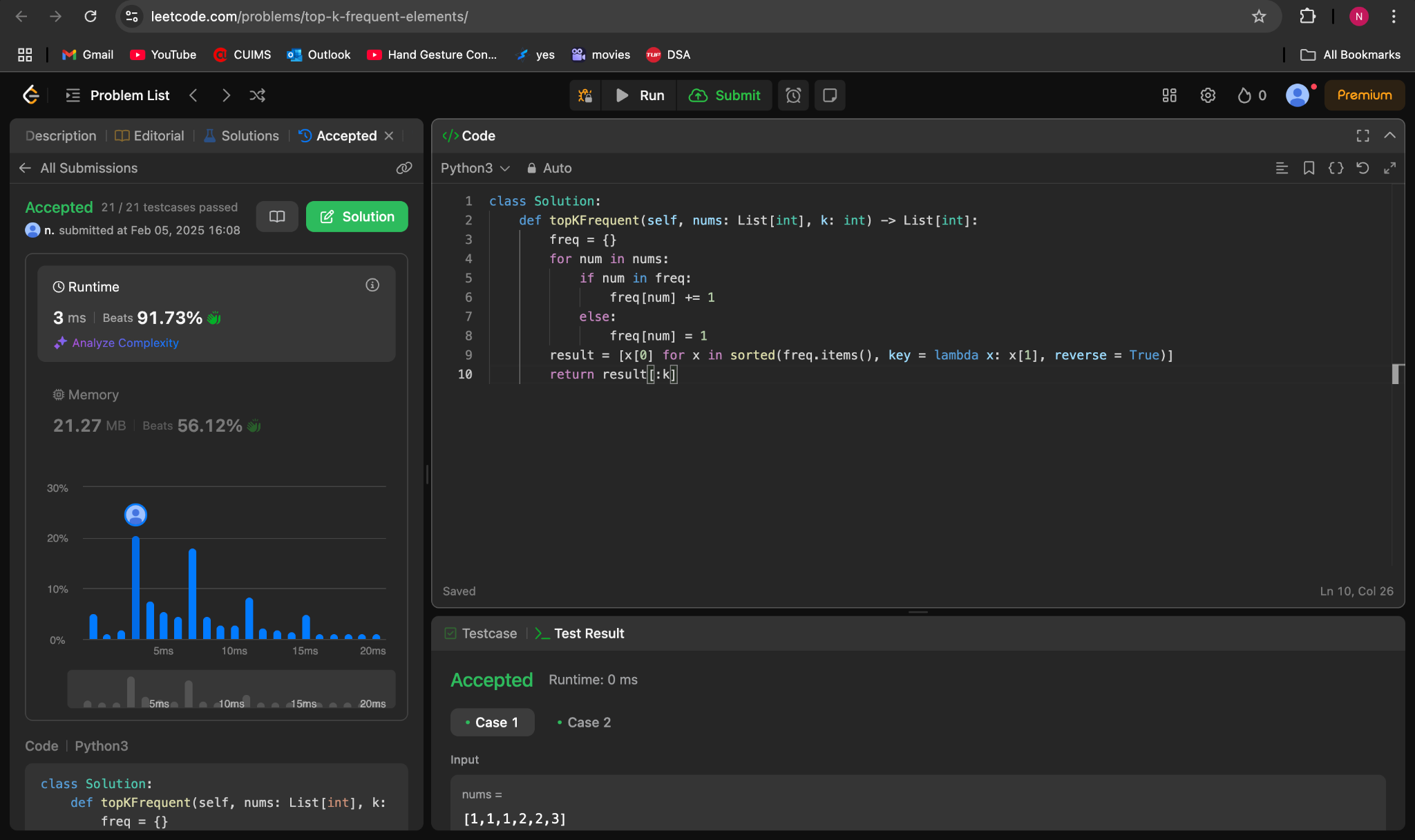
**freq[num] += 1**

**else:**

**freq[num] = 1**

**result = [x[0] for x in sorted(freq.items(), key = lambda x: x[1], reverse = True)]**

**return result[:k]**

****

**215.Kth Largest Element in an Array**

**class Solution:**

**def findKthLargest(self, nums: List[int], k: int) -> int:**

**nums=sorted(nums)[::-1]**

**return nums[k-1]**

****

**162.Find Peak Element**

**class Solution:**

**def findPeakElement(self, nums: List[int]) -> int:**

**l, r = 0, len(nums) - 1**

**while l < r:**

**mid = (l + r) // 2**

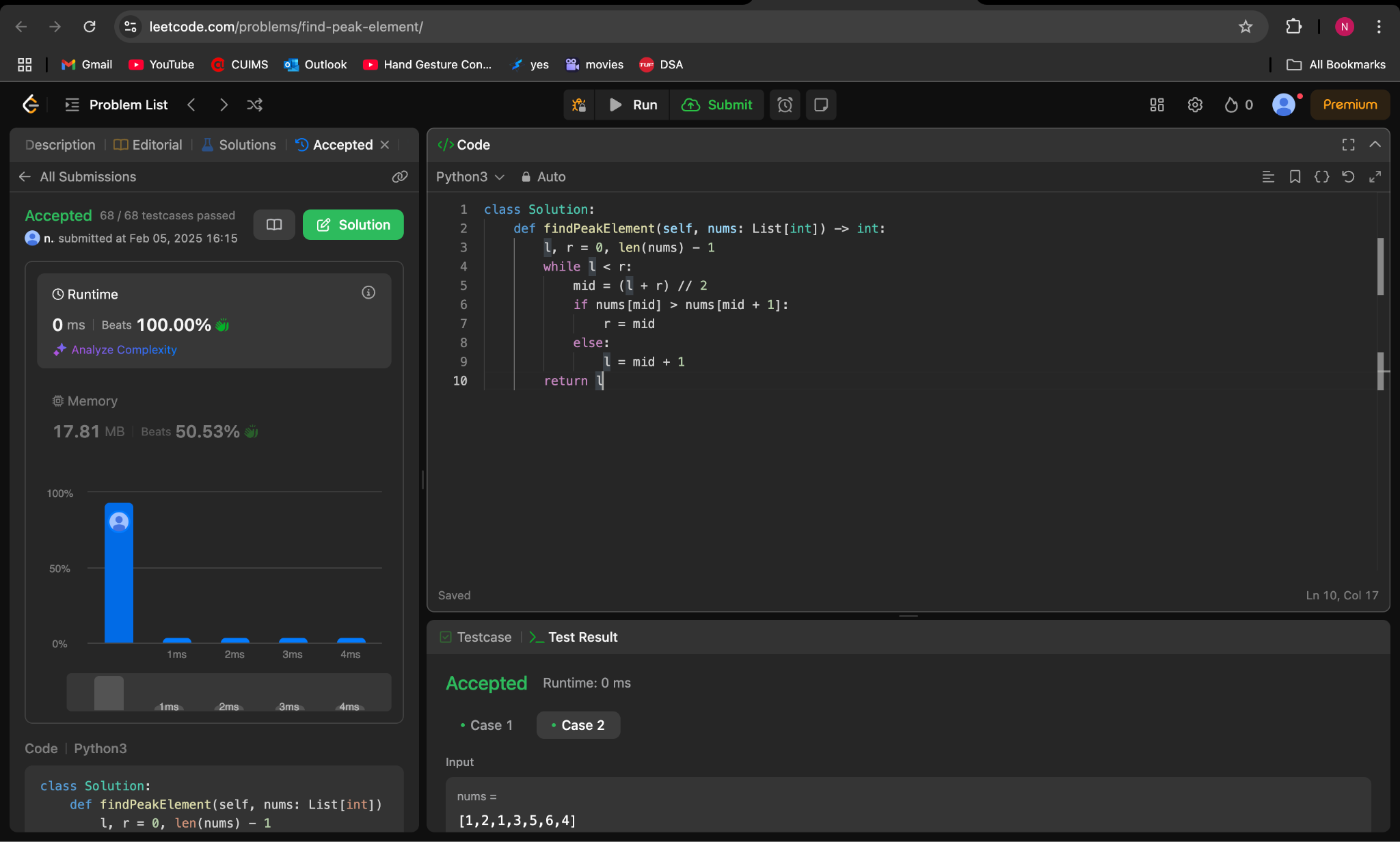
**if nums[mid] > nums[mid + 1]:**

**r = mid**

**else:**

**l = mid + 1**

**return l**

****

**56.Merge Intervals**

**class Solution:**

**def merge(self, intervals: List[List[int]]) -> List[List[int]]:**

**merged = []**

**intervals.sort(key=lambda x: x[0])**

**prev = intervals[0]**

**for interval in intervals[1:]:**

**if interval[0] <= prev[1]:**

**prev[1] = max(prev[1], interval[1])**

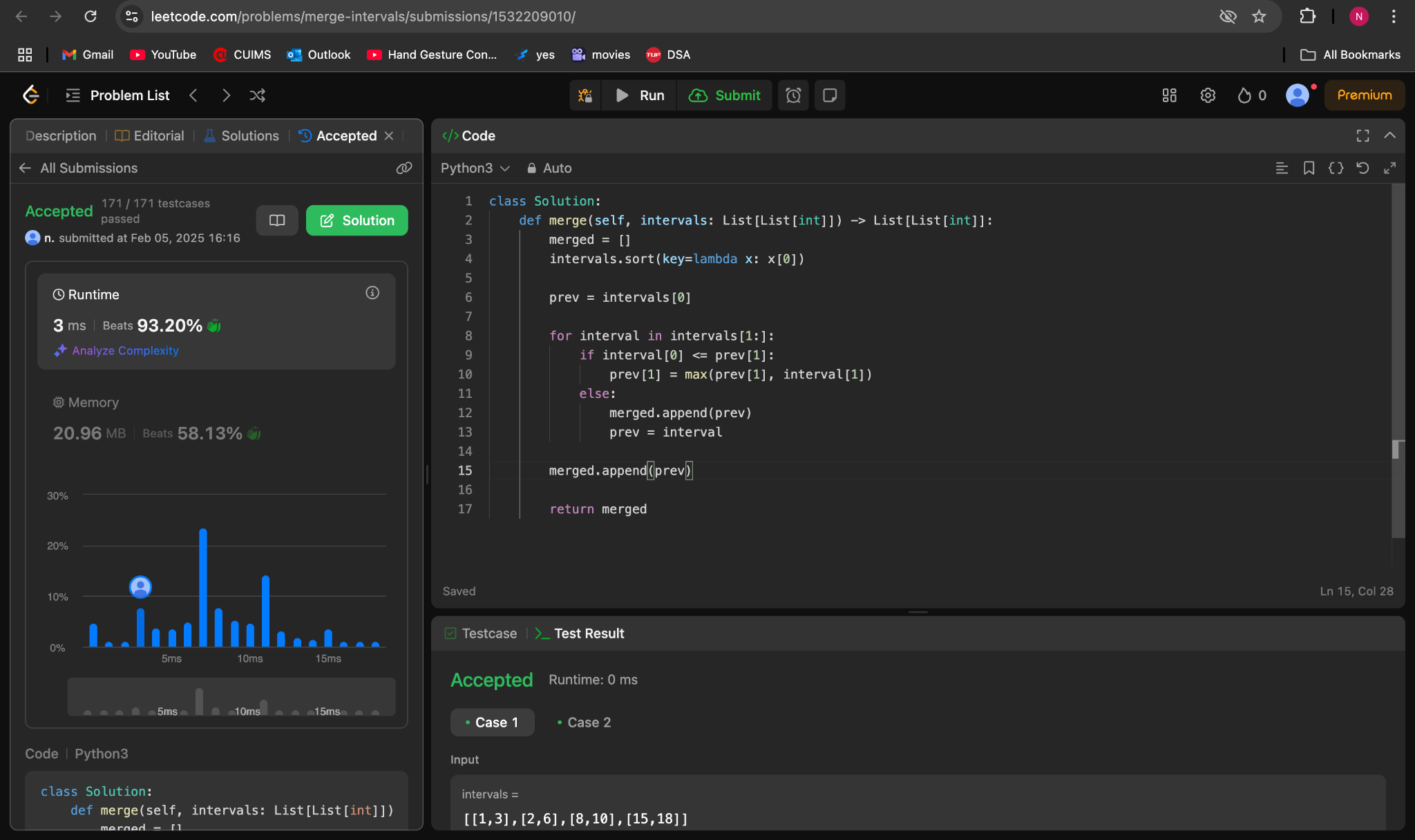
**else:**

**merged.append(prev)**

**prev = interval**

**merged.append(prev)**

**return merged**

****

**33.Search in Rotated Sorted Array**

**class Solution:**

**def search(self, nums: List[int], target: int) -> int:**

**left = 0**

**right = len(nums) - 1**

**while left <= right:**

**mid = (left + right) // 2**

**if nums[mid] == target:**

**return mid**

**elif nums[mid] >= nums[left]:**

**if nums[left] <= target <= nums[mid]:**

**right = mid - 1**

**else:**

**left = mid + 1**

**else:**

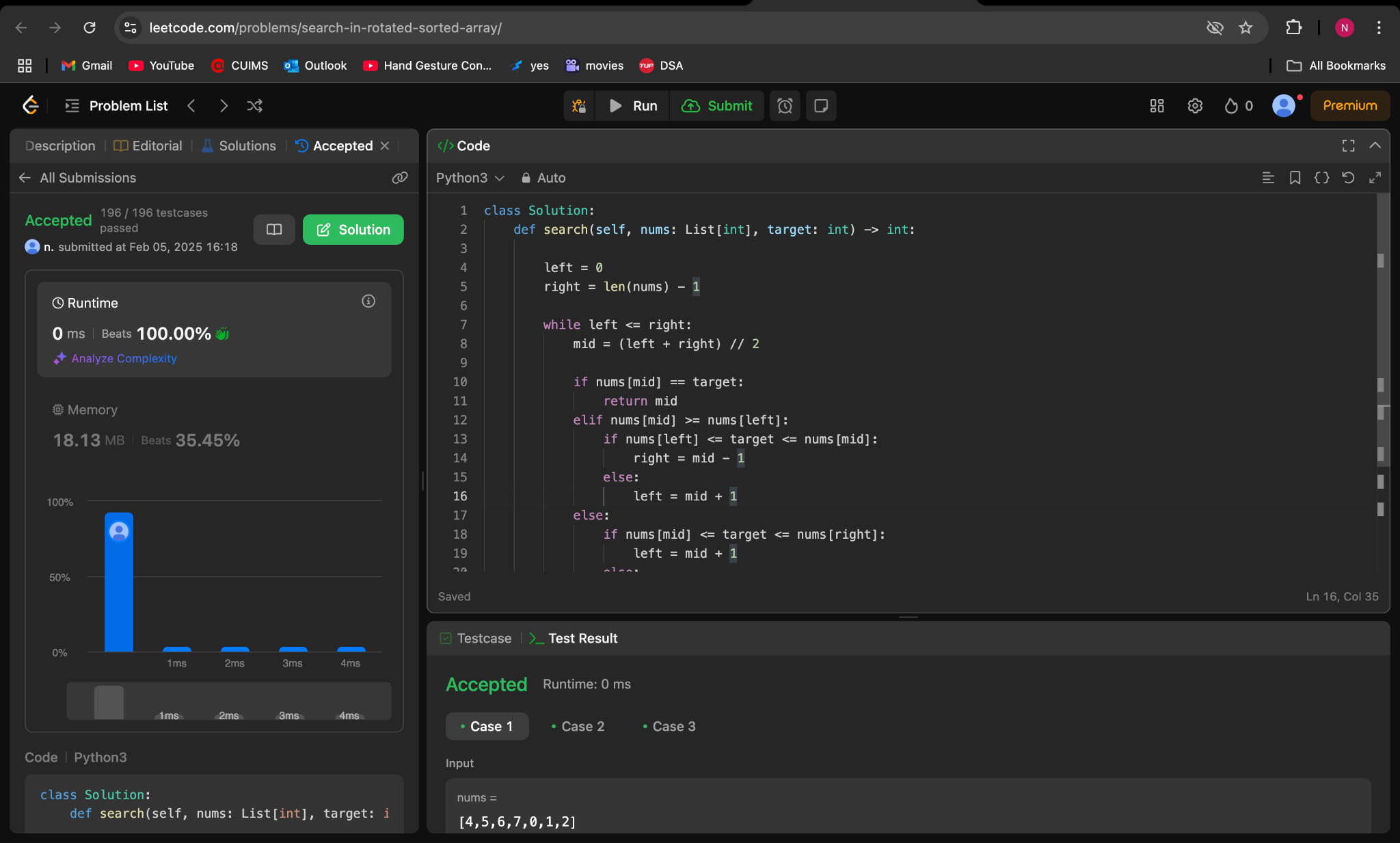
**if nums[mid] <= target <= nums[right]:**

**left = mid + 1**

**else:**

**right = mid - 1**

**return -1**

****

**324.Wiggle Sort II**

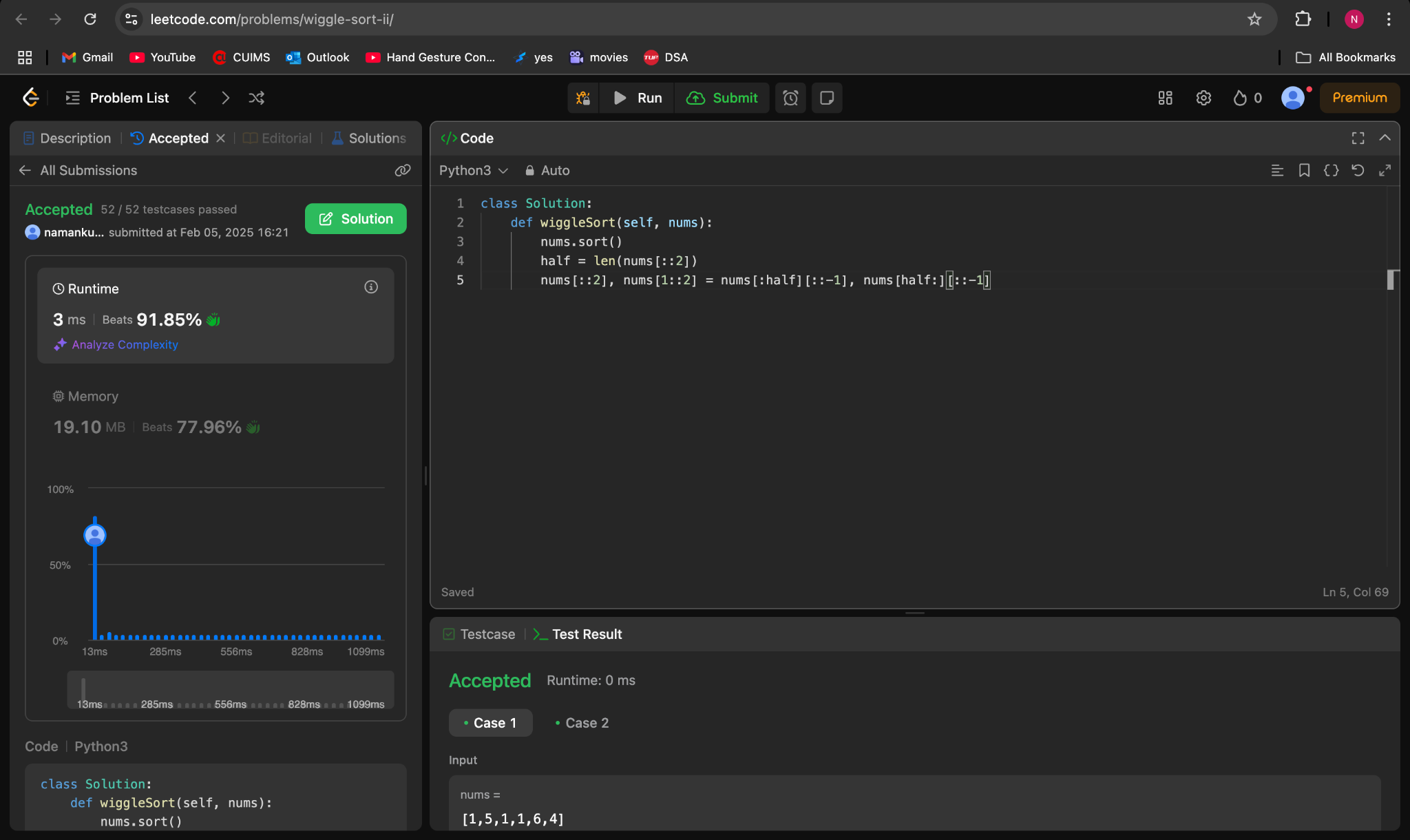
**class Solution:**

**def wiggleSort(self, nums):**

**nums.sort()**

**half = len(nums[::2])**

**nums[::2], nums[1::2] = nums[:half][::-1], nums[half:][::-1]**

****

**378.Kth Smallest Element in a Sorted Matrix**

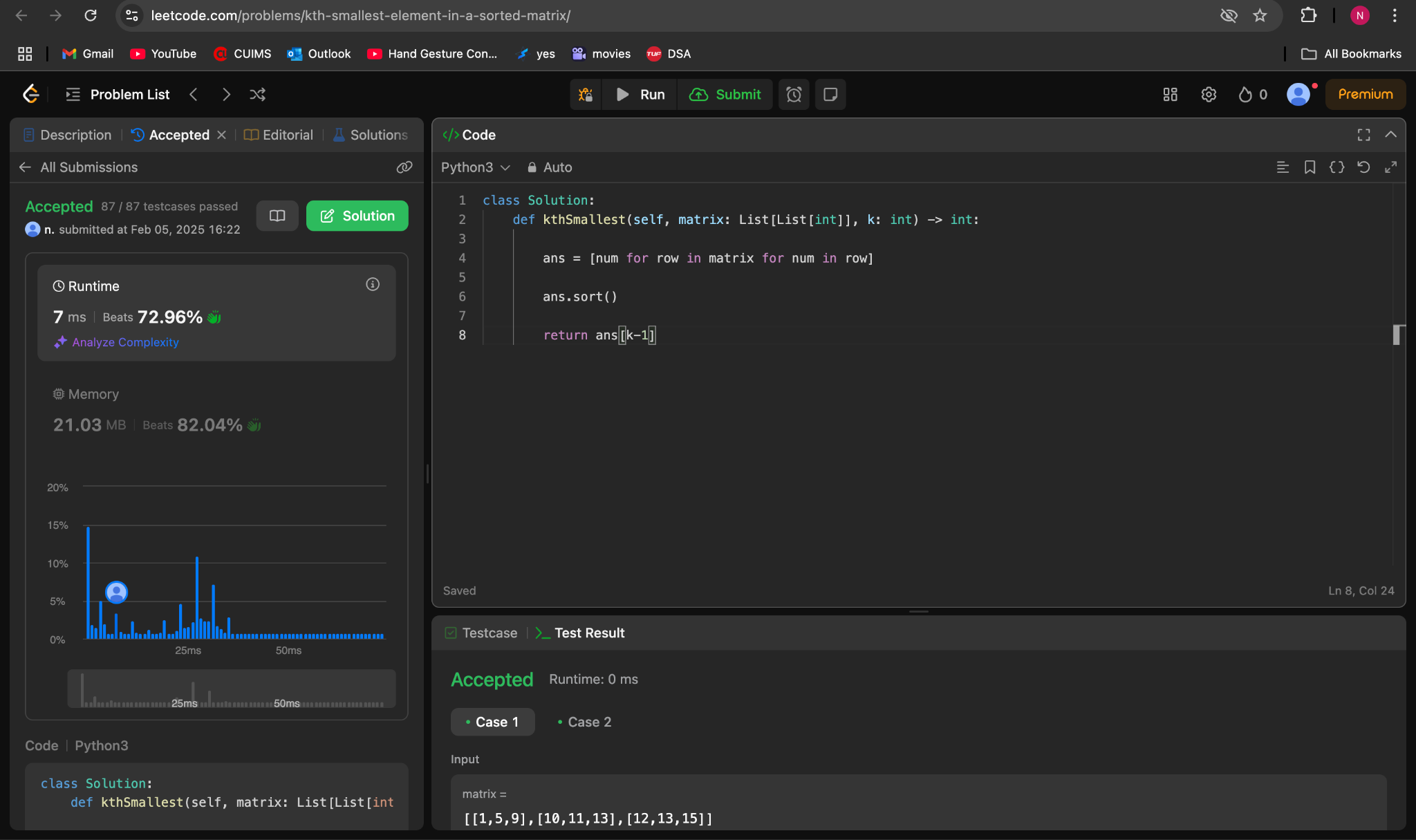
**class Solution:**

**def kthSmallest(self, matrix: List[List[int]], k: int) -> int:**

**ans = [num for row in matrix for num in row]**

**ans.sort()**

**return ans[k-1]**

****

**4.Median of Two Sorted Arrays**

**class Solution(object):**

**def findMedianSortedArrays(self, nums1, nums2):**

**if len(nums1) > len(nums2):**

**nums1, nums2 = nums2, nums1**

**NUMS1\_LENGTH = len(nums1)**

**NUMS2\_LENGTH = len(nums2)**

**TOTAL\_LENGTH = NUMS1\_LENGTH + NUMS2\_LENGTH**

**HALF\_TOTAL\_LENGTH = TOTAL\_LENGTH // 2**

**left = 0**

**right = NUMS1\_LENGTH - 1**

**while True:**

**middle1 = (left + right) // 2**

**middle2 = HALF\_TOTAL\_LENGTH - middle1 - 2**

**nums1left = nums1[middle1] if middle1 >= 0 else float("-infinity")**

**nums1right = nums1[middle1+1] if middle1+1 < NUMS1\_LENGTH else float("infinity")**

**nums2left = nums2[middle2] if middle2 >= 0 else float("-infinity")**

**nums2right = nums2[middle2+1] if middle2+1 < NUMS2\_LENGTH else float("infinity")**

**if nums1left <= nums2right and nums2left <= nums1right:**

**if TOTAL\_LENGTH % 2 == 0:**

**return (max(nums1left, nums2left) + min(nums1right, nums2right)) / 2.0**

**else:**

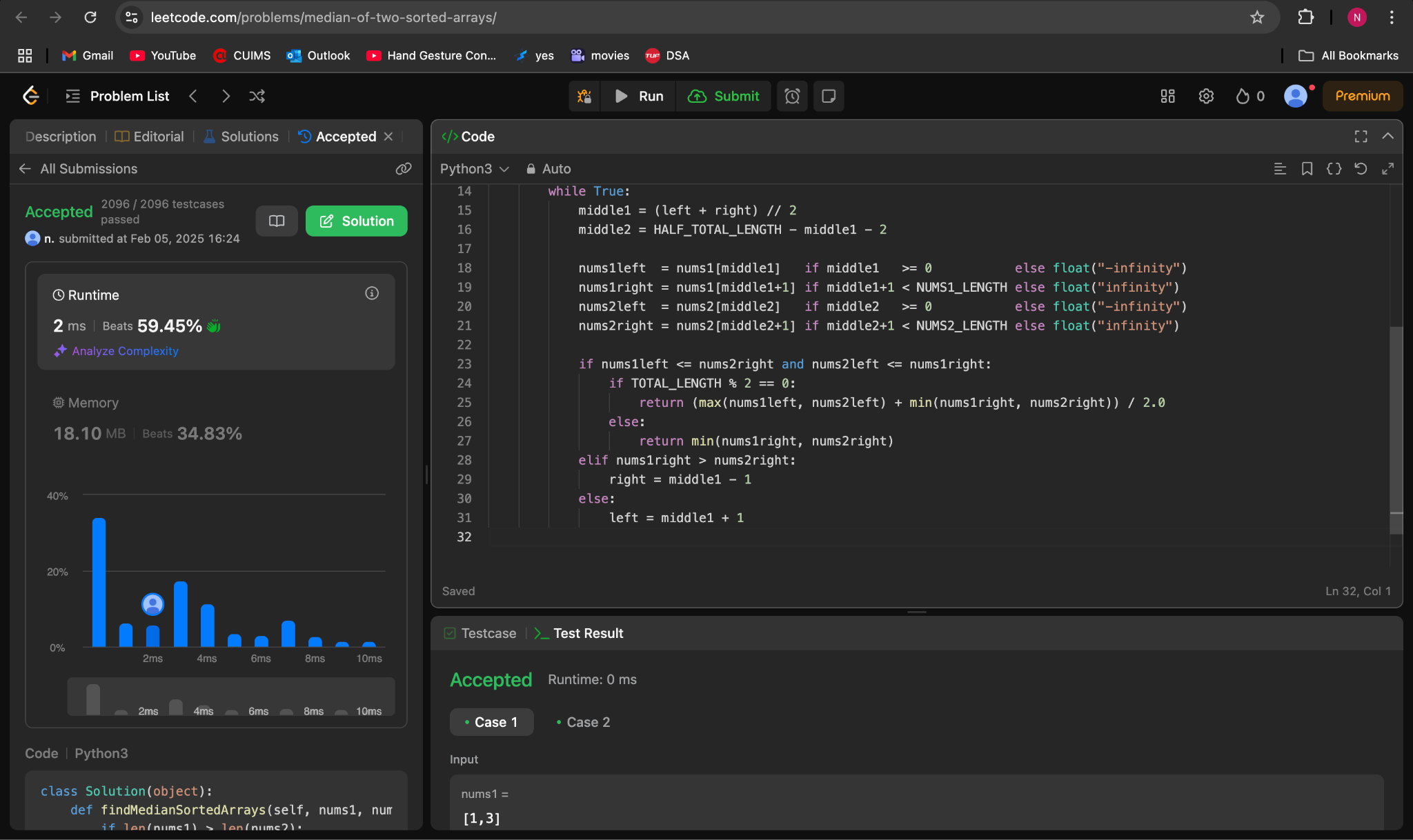
**return min(nums1right, nums2right)**

**elif nums1right > nums2right:**

**right = middle1 - 1**

**else:**

**left = middle1 + 1**

****