1.Given an integer array nums, return true if any value appears **at least twice** in the array, and return false if every element is distinct.

class Solution {

    public boolean containsDuplicate(int[] nums) {

        HashSet<Integer> seen = new HashSet<>();

        for (int num : nums) {

            if (seen.contains(num))

                return true;

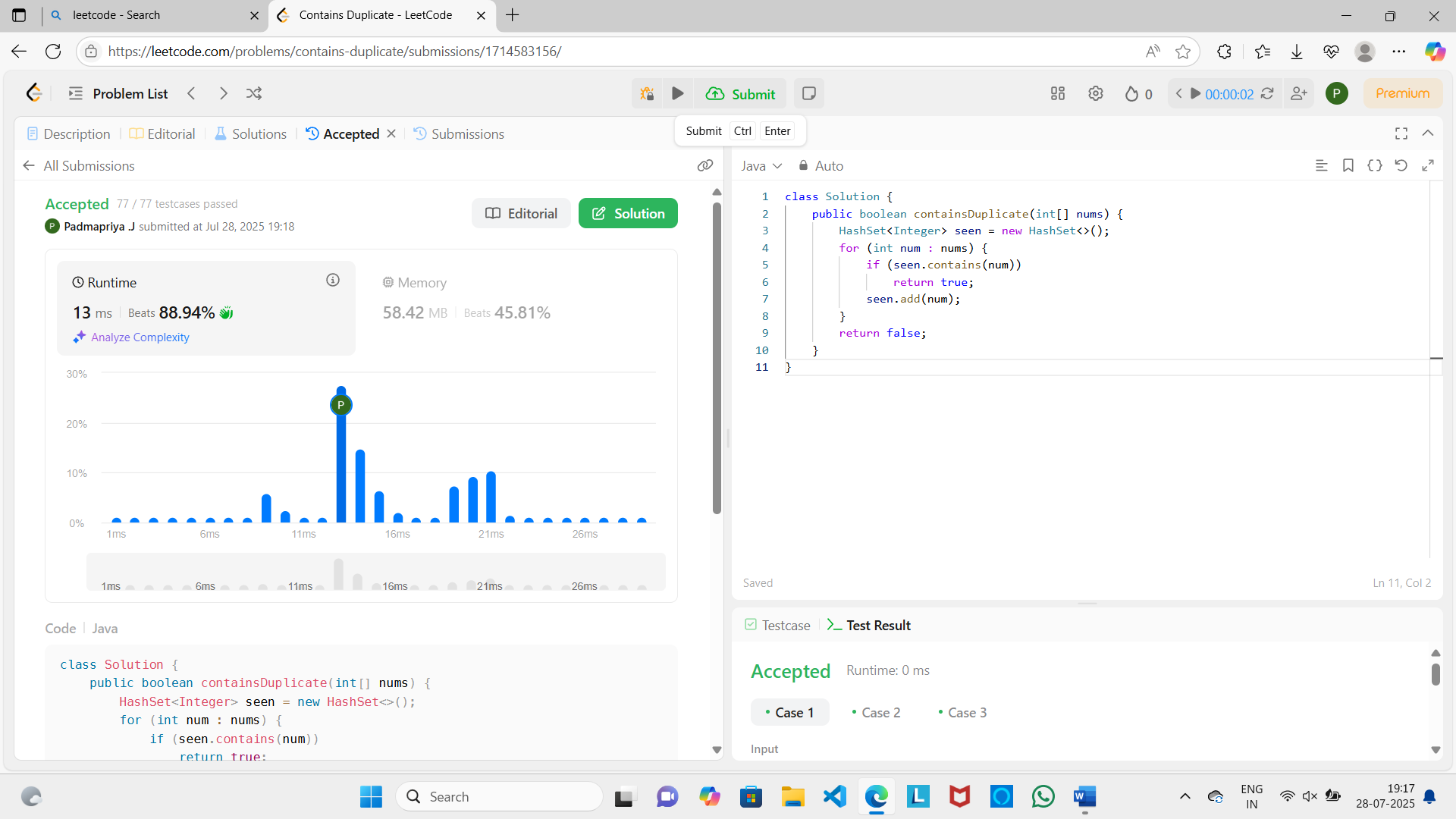
            seen.add(num);

        }

        return false;

    }

}



2. Given a **non-empty** array of integers nums, every element appears *twice* except for one. Find that single one.

You must implement a solution with a linear runtime complexity and use only constant extra space.

class Solution {

public int singleNumber(int[] nums) {

int c=0;

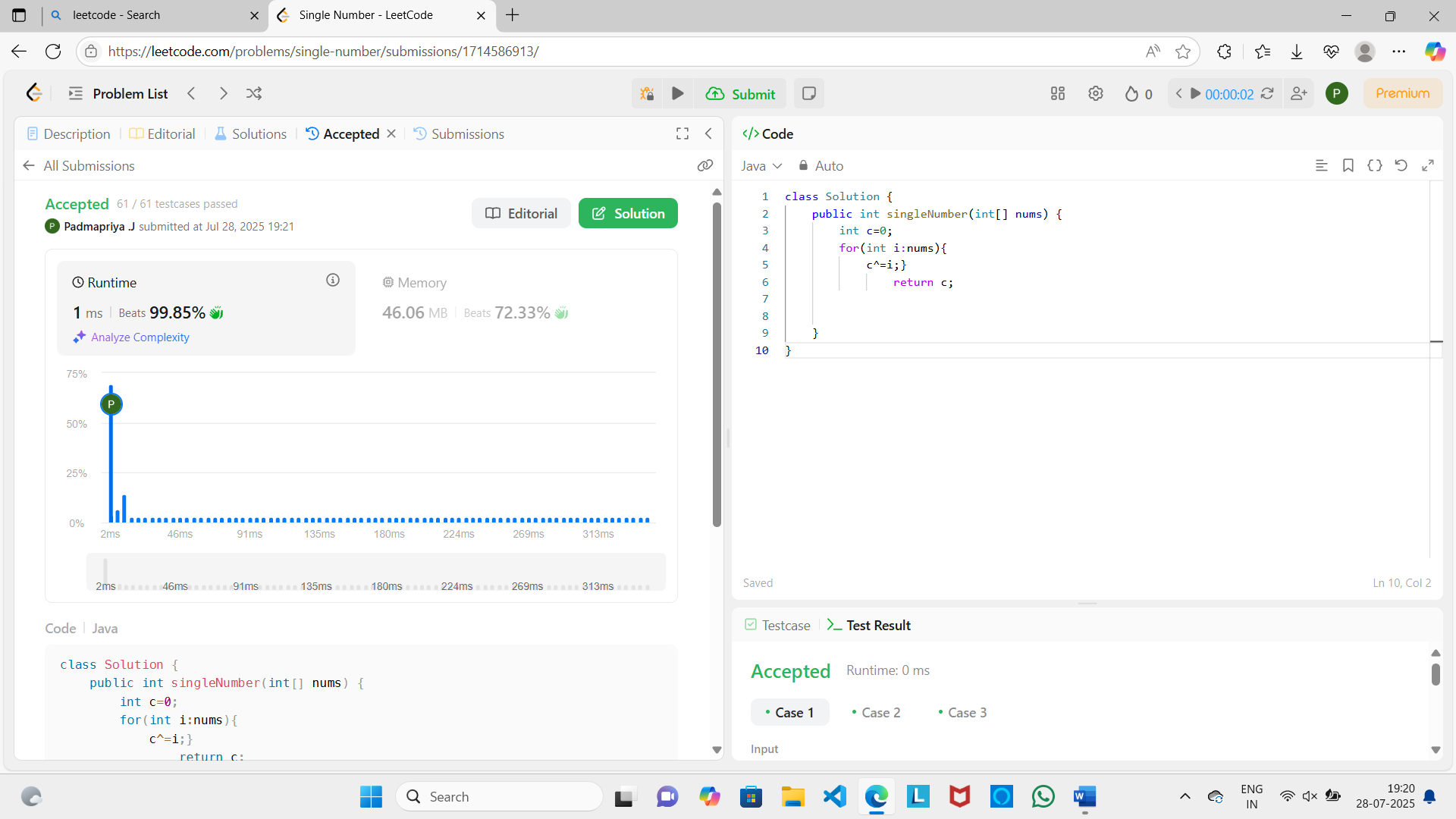
for(int i:nums){

c^=i;}

return c;

}

}



3. You are given two integer arrays nums1 and nums2, sorted in **non-decreasing order**, and two integers m and n, representing the number of elements in nums1 and nums2 respectively.

class Solution {

    public void merge(int[] nums1, int m, int[] nums2, int n) {

        for (int j = 0, i = m; j < n; j++) {

            nums1[i] = nums2[j];

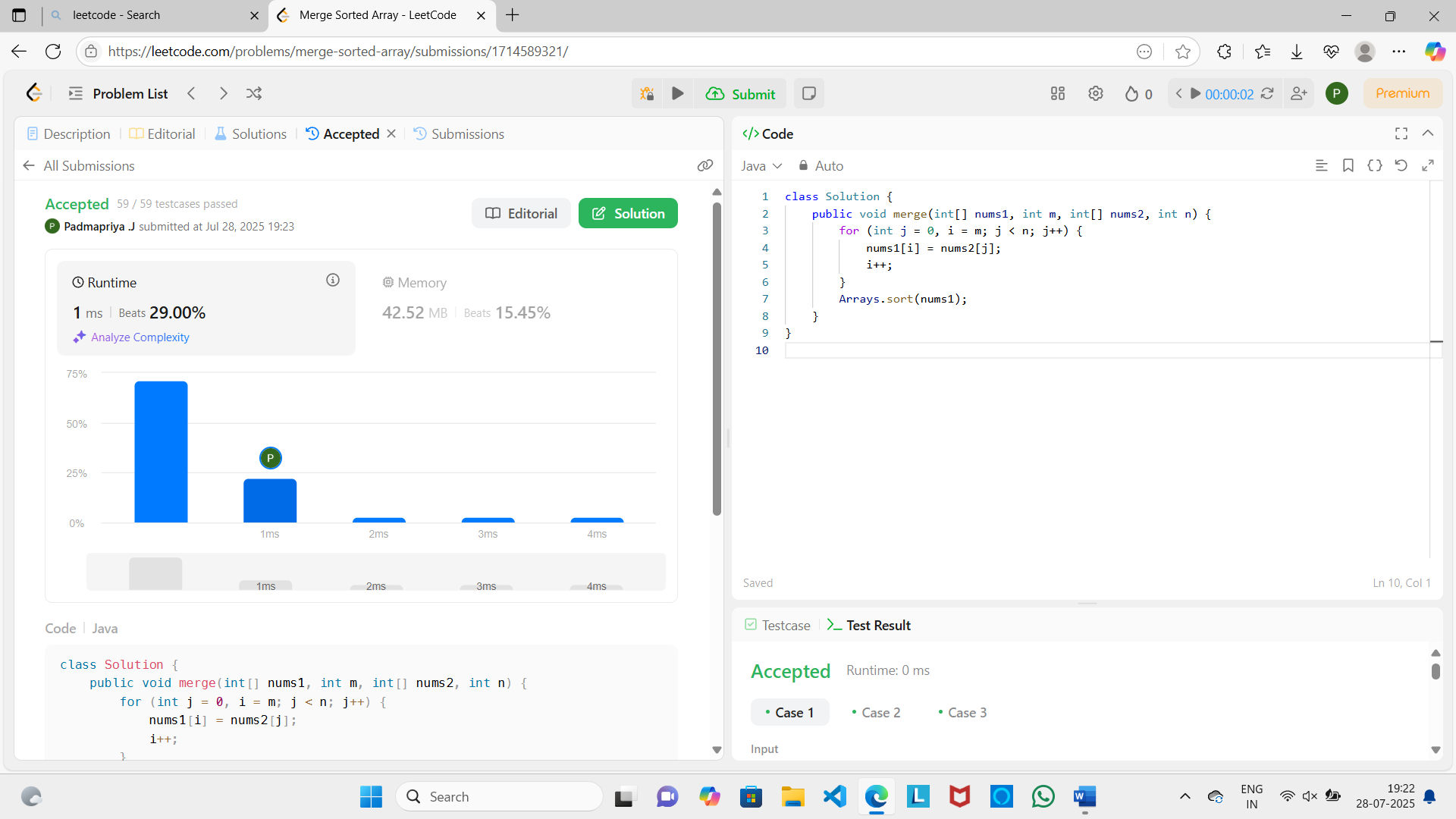
            i++;

        }

        Arrays.sort(nums1);

    }

}



4. Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

class Solution {

public int searchInsert(int[] nums, int target) {

int arr[]=new int[nums.length+1];

int i=0;

for(i=0;i<nums.length;i++){

arr[i]=nums[i];

}

arr[i]=target;

Arrays.sort(arr);

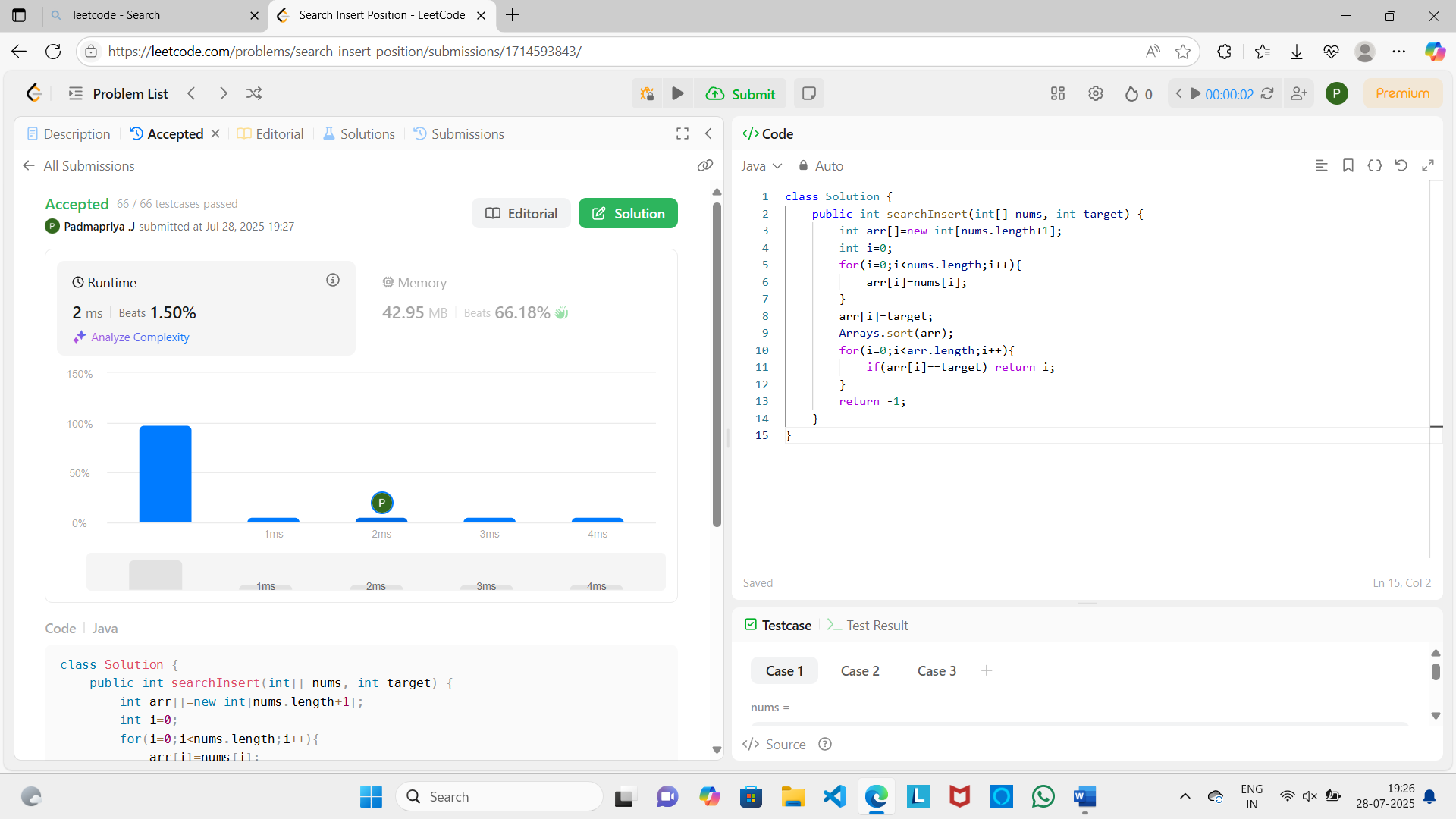
for(i=0;i<arr.length;i++){

if(arr[i]==target) return i;

}

return -1;

}}



5. Given an integer array nums sorted in **non-decreasing order**, remove the duplicates [**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm) such that each unique element appears only **once**. The **relative order** of the elements should be kept the **same**. Then return *the number of unique elements in*nums.

class Solution {

    public int removeDuplicates(int[] nums) {

        int j = 1;

        for (int i = 1; i < nums.length; i++) {

            if (nums[i] != nums[i - 1]) {

                nums[j] = nums[i];

                j++;

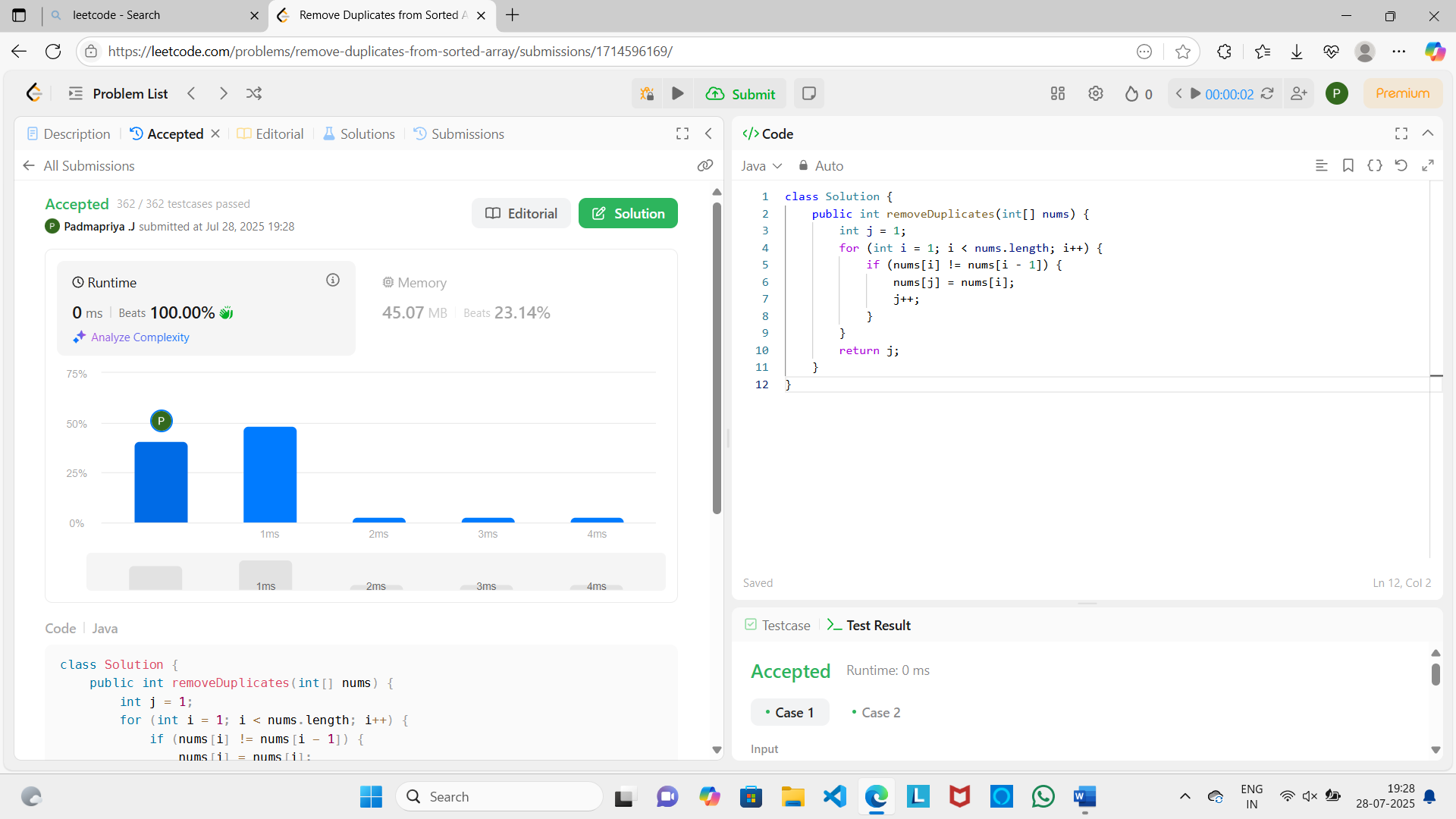
            }

        }

        return j;

    }

}



6. Given two strings s and t, return true*if*s*is a****subsequence****of*t*, or*false*otherwise*.

A **subsequence** of a string is a new string that is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (i.e., "ace" is a subsequence of "abcde" while "aec" is not).

class Solution {

public boolean isSubsequence(String s, String t) {

int i = 0;

int j = 0;

while (i < s.length() && j < t.length()) {

if (s.charAt(i) == t.charAt(j)) {

i++;

}

j++;

}

return i == s.length();

}

}

