**Advanced Programming-2**

**Assignment-2**

**Q1. Longest Nice Substring.**

Code:

class Solution {

public String longestNiceSubstring(String s) {

String output = "";

int count = 0;

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

int smallMask = 0;

int largeMask = 0;

for (int j = i; j < s.length(); j++) {

char ch = s.charAt(j);

int chint;

if (Character.isUpperCase(ch)) {

chint = ch - 'A';

largeMask |= (1 << chint);

} else {

chint = ch - 'a';

smallMask |= (1 << chint);

}

if ((smallMask ^ largeMask) == 0) {

if (count < j - i + 1) {

count = j - i + 1;

output = s.substring(i, j + 1);

}

}

}

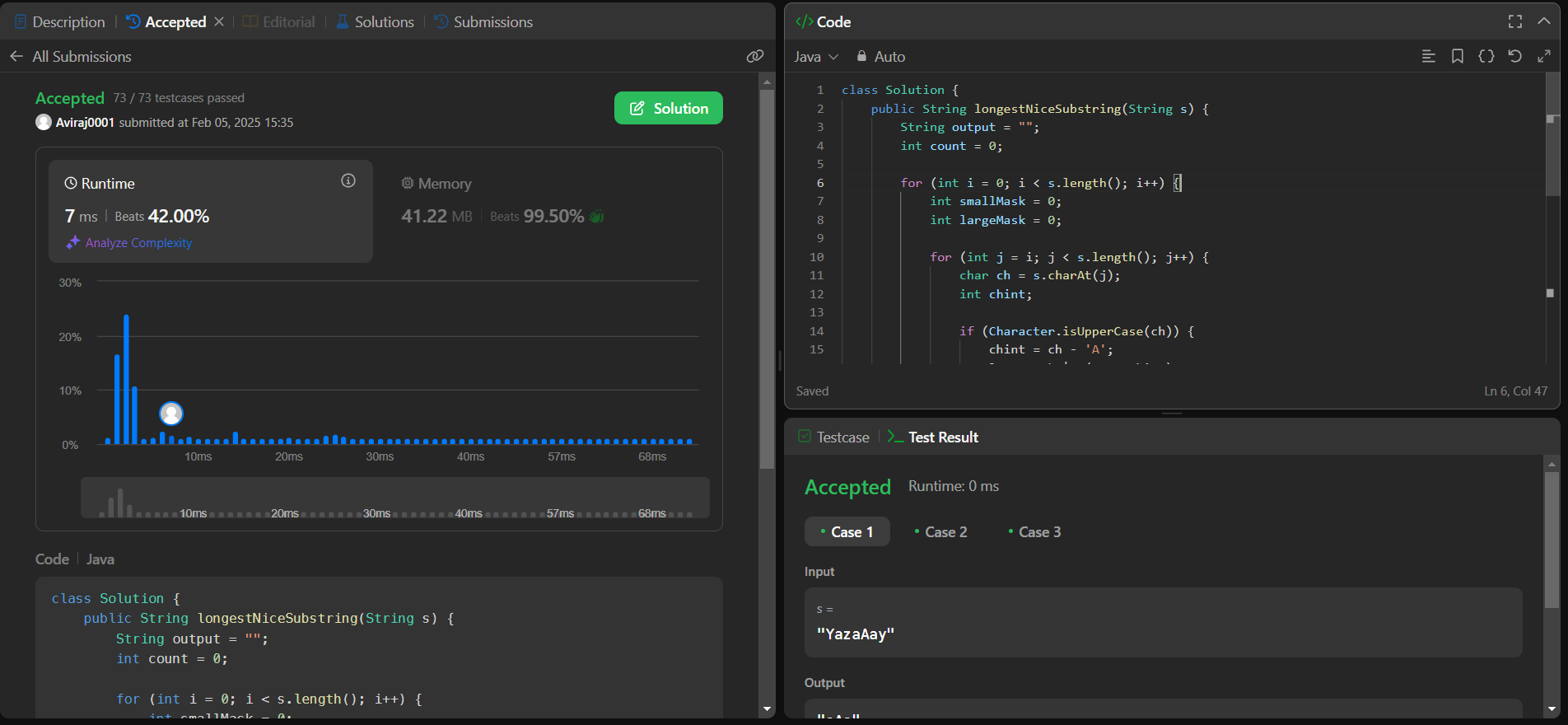
}

return output;

}

}

**Output:**

****

**Q. Maximum Subarray.**

Code:

class Solution {

public int maxSubArray(int[] nums) {

int maxSum = nums[0];

int currentSum = nums[0];

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

currentSum = Math.max(nums[i], currentSum + nums[i]);

maxSum = Math.max(maxSum, currentSum);

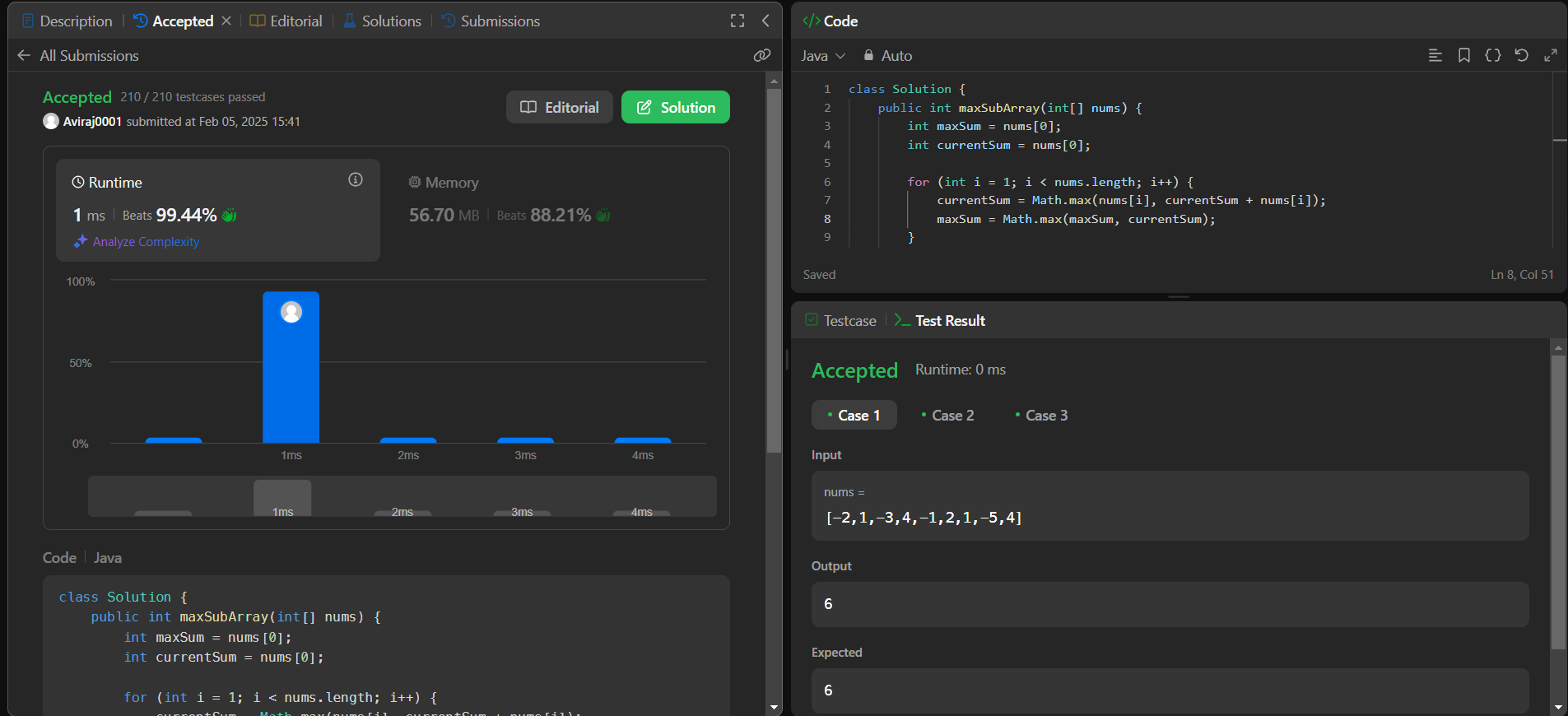
}

return maxSum;

}

}

**Output:**

****

**Q. Reverese Bits.**

**Code:**

public class Solution {

public int reverseBits(int n) {

int result = 0;

for (int i = 0; i < 32; i++) {

result <<= 1;

result |= (n & 1);

n >>= 1;

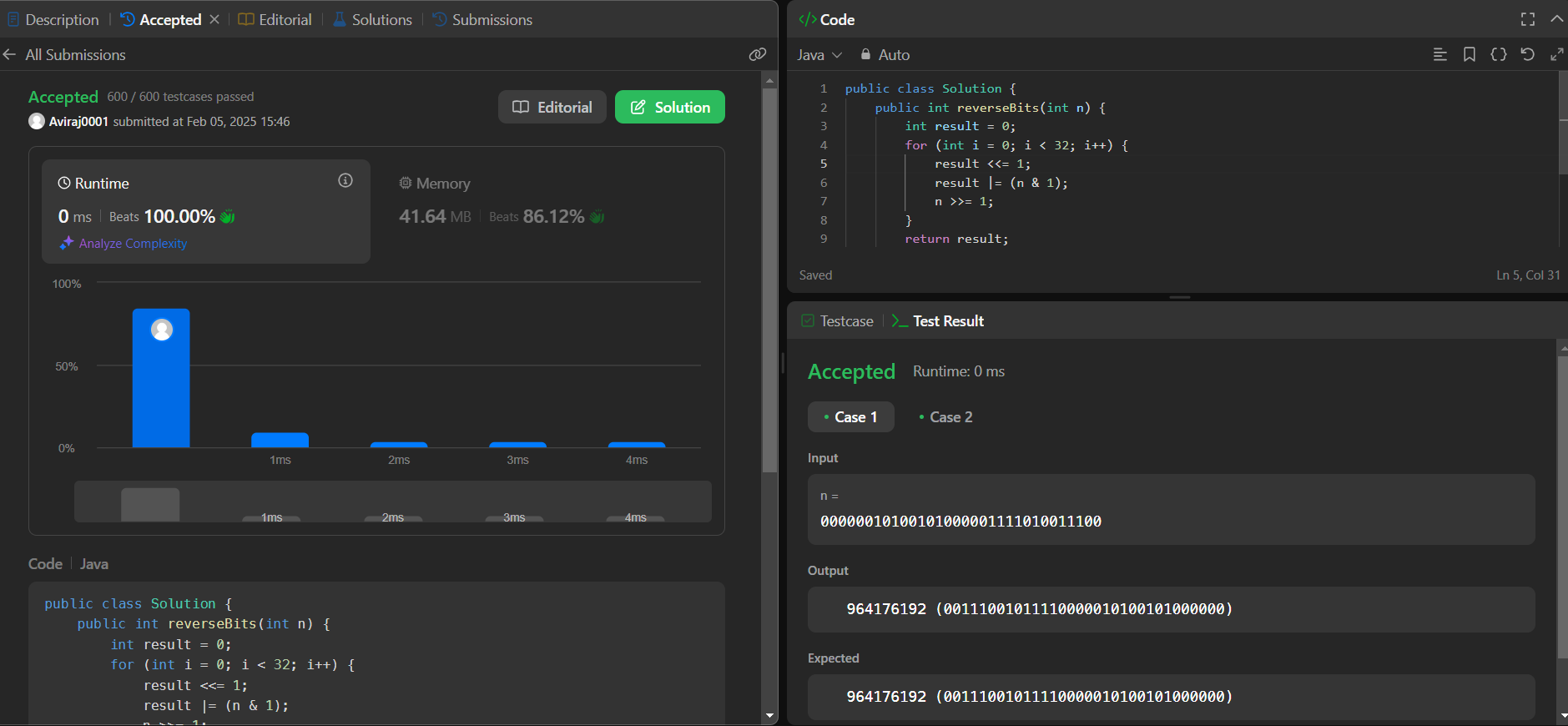
}

return result;

}

}

**Output:**

****

**Q. Number of 1 bits.**

**Code:**

class Solution {

public int hammingWeight(int n) {

int count = 0;

while (n != 0) {

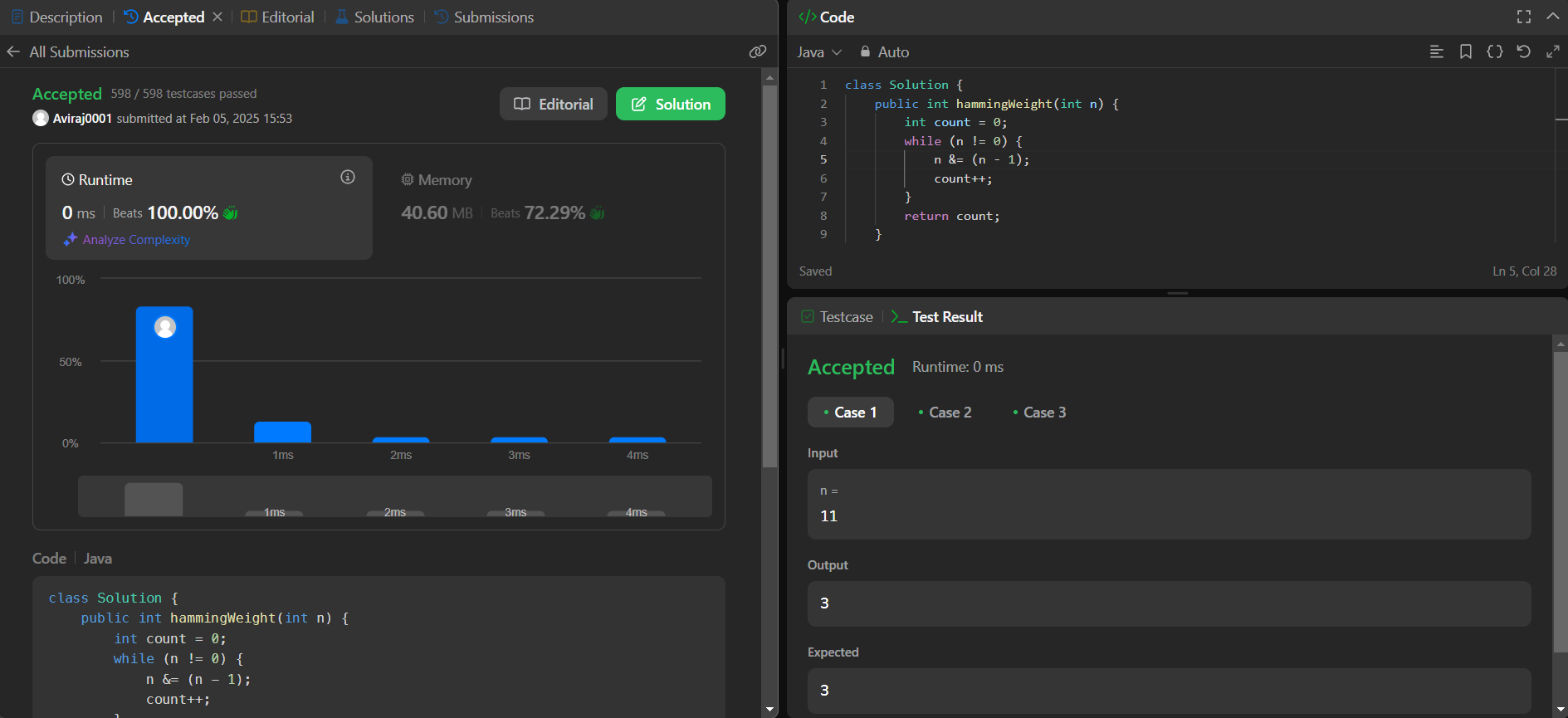
n &= (n - 1);

count++; }

return count;

}}

**Output:**

****

**Q. Merge Two Sorted Arrays.**

**Code:**

class Solution {

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

int i = m - 1;

int j = n - 1;

int k = m + n - 1;

while (i >= 0 && j >= 0) {

if (nums1[i] > nums2[j]) {

nums1[k--] = nums1[i--];

} else {

nums1[k--] = nums2[j--];

}}

while (j >= 0) {

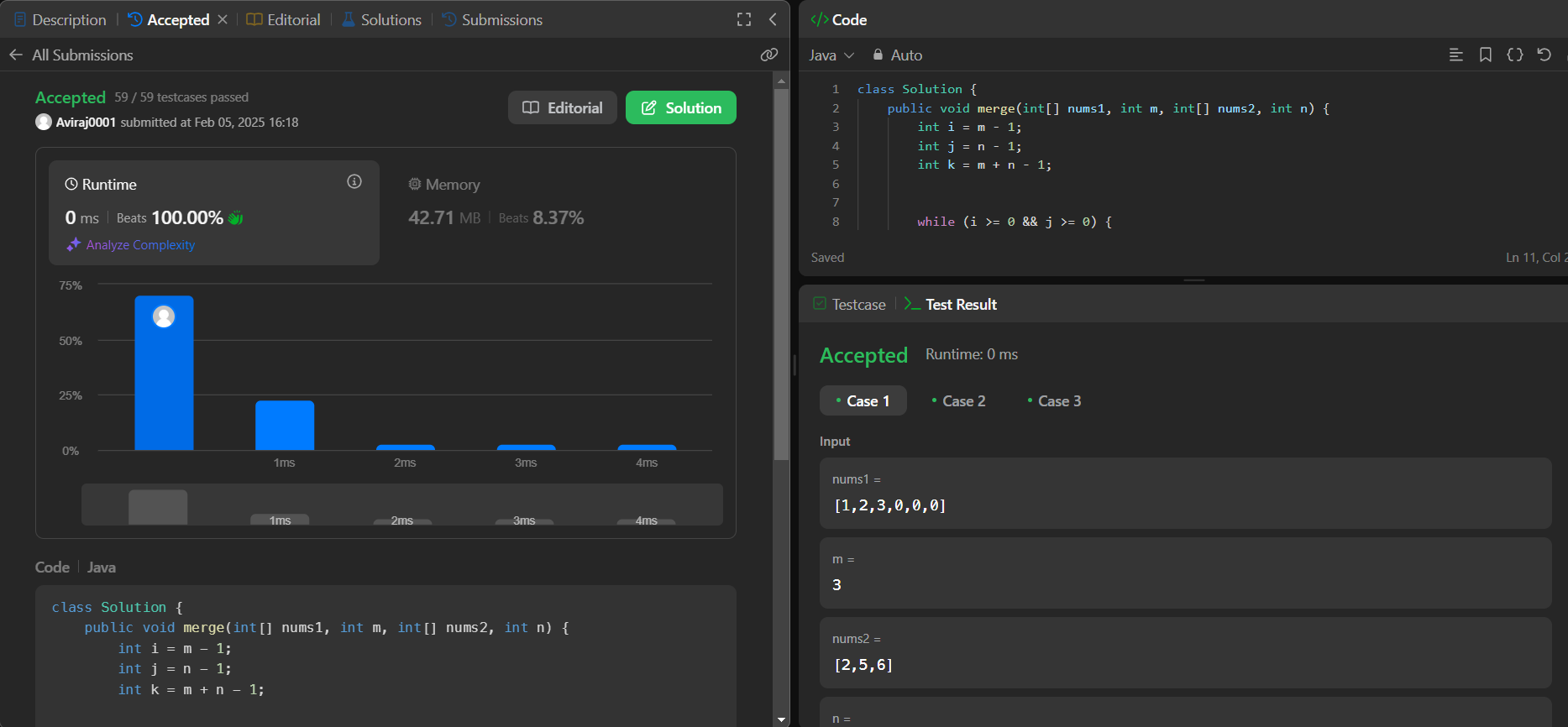
nums1[k--] = nums2[j--];

}

}

}

**Output:**

****

**Q. Search a 2D Matrix.**

**Code:**

class Solution {

public boolean searchMatrix(int[][] matrix, int target) {

int m = matrix.length;

int n = matrix[0].length;

int row = 0, col = n - 1;

while (row < m && col >= 0) {

if (matrix[row][col] == target) {

return true;

} else if (matrix[row][col] > target) {

col--;

} else {

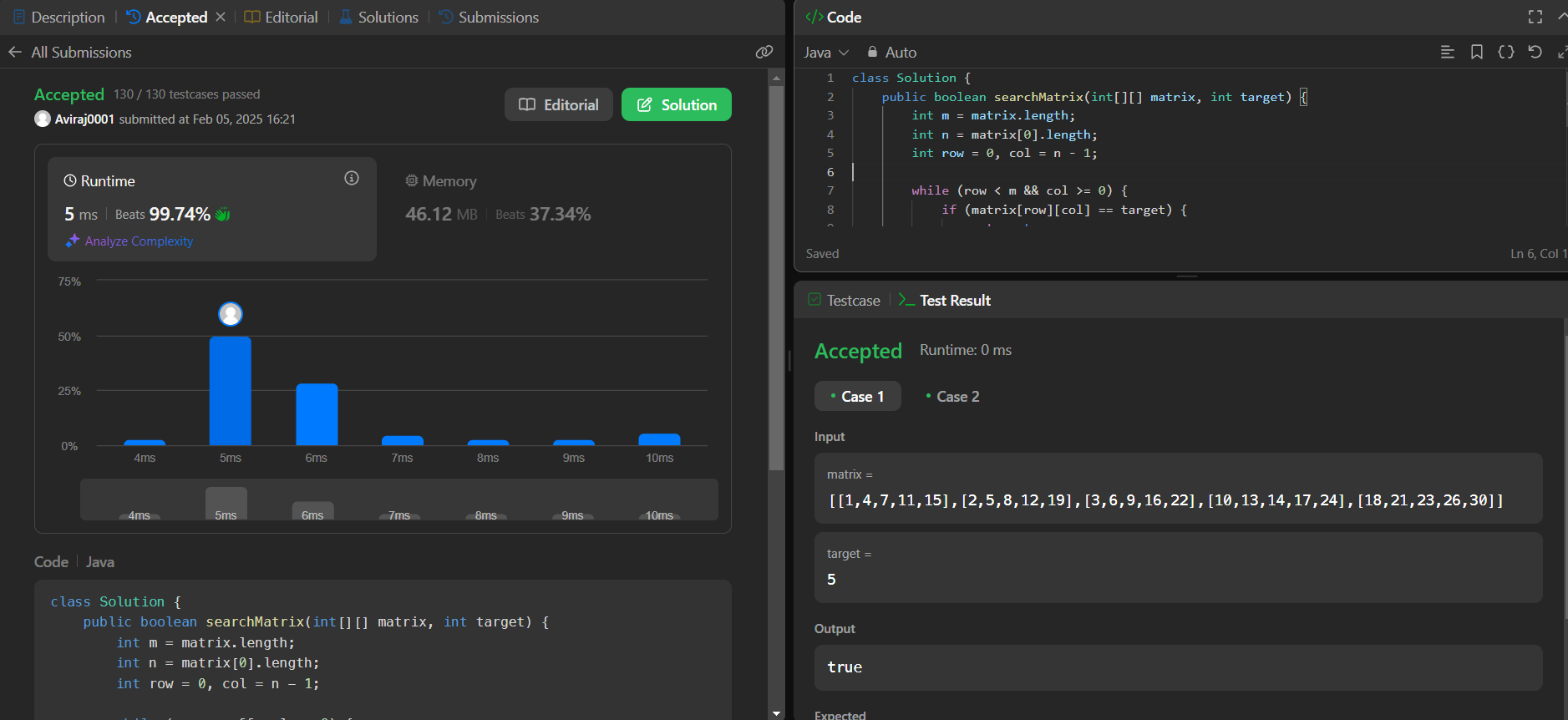
row++;

} }

return false;

}}

**Output:**

****

**Q. Super Pow.**

**Code:**

class Solution {

private static final int MOD = 1337;

private int pow(int a, int b) {

int result = 1;

a %= MOD; // Taking mod to prevent overflow

for (int i = 0; i < b; i++) {

result = (result \* a) % MOD;

}

return result;

}

public int superPow(int a, int[] b) {

int result = 1;

for (int i = b.length - 1; i >= 0; i--) {

result = (result \* pow(a, b[i])) % MOD;

a = pow(a, 10); // Power up for the next iteration

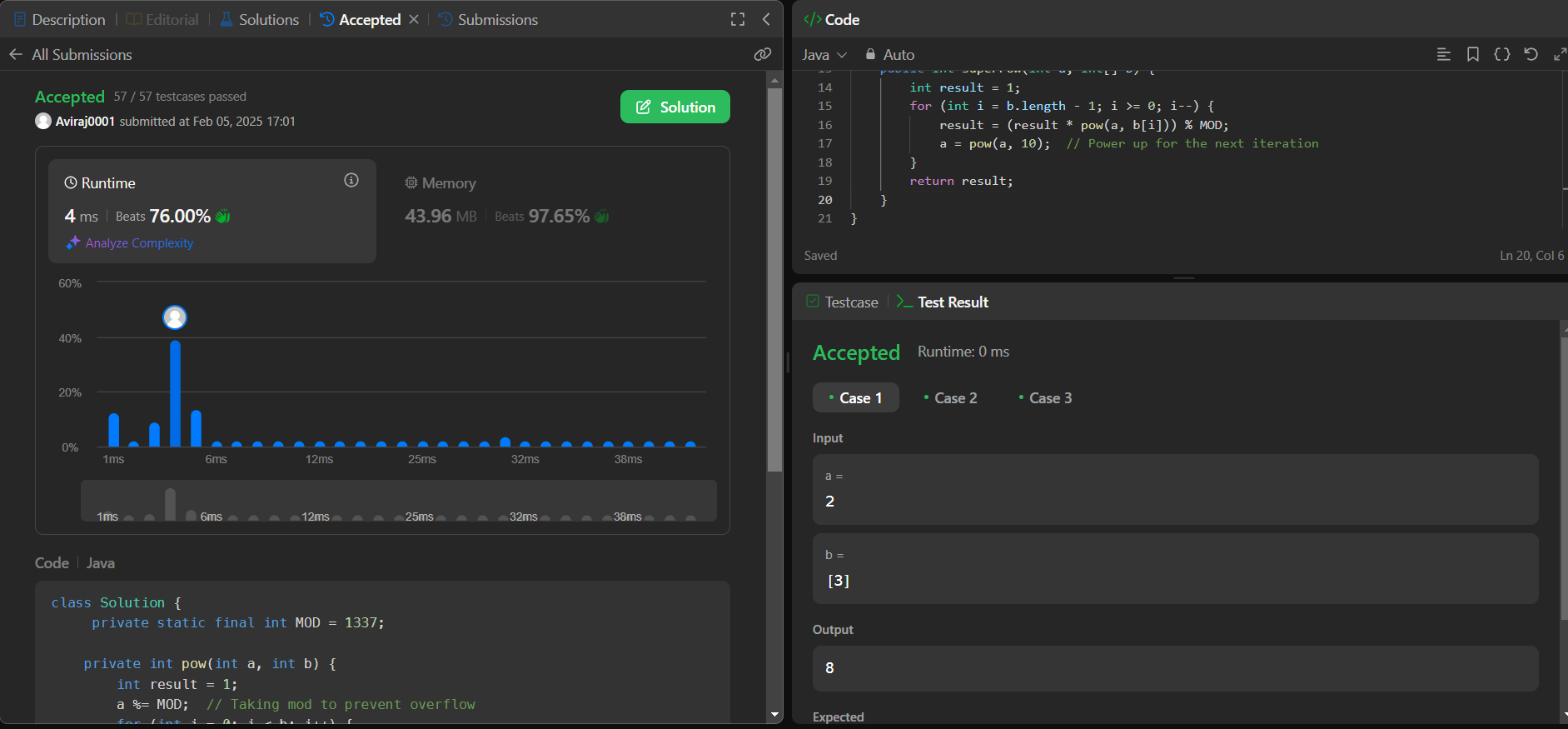
}

return result;

}

}

**Output:**

****

**Q. Beautiful Array.**

**Code:**

class Solution {

public int[] beautifulArray(int N) {

int[] res = new int[N];

if (N == 1)

{

return new int[] {1};

}

else if (N == 2)

{

return new int[] {1, 2};

}

else

{

int[] odds = beautifulArray((N + 1) / 2);

int[] even = beautifulArray(N / 2);

for (int i = 0; i < odds.length; i ++)

{

res[i] = odds[i] \* 2 - 1;

}

for (int j = 0; j < even.length; j ++)

{

res[odds.length + j] = even[j] \* 2;

}

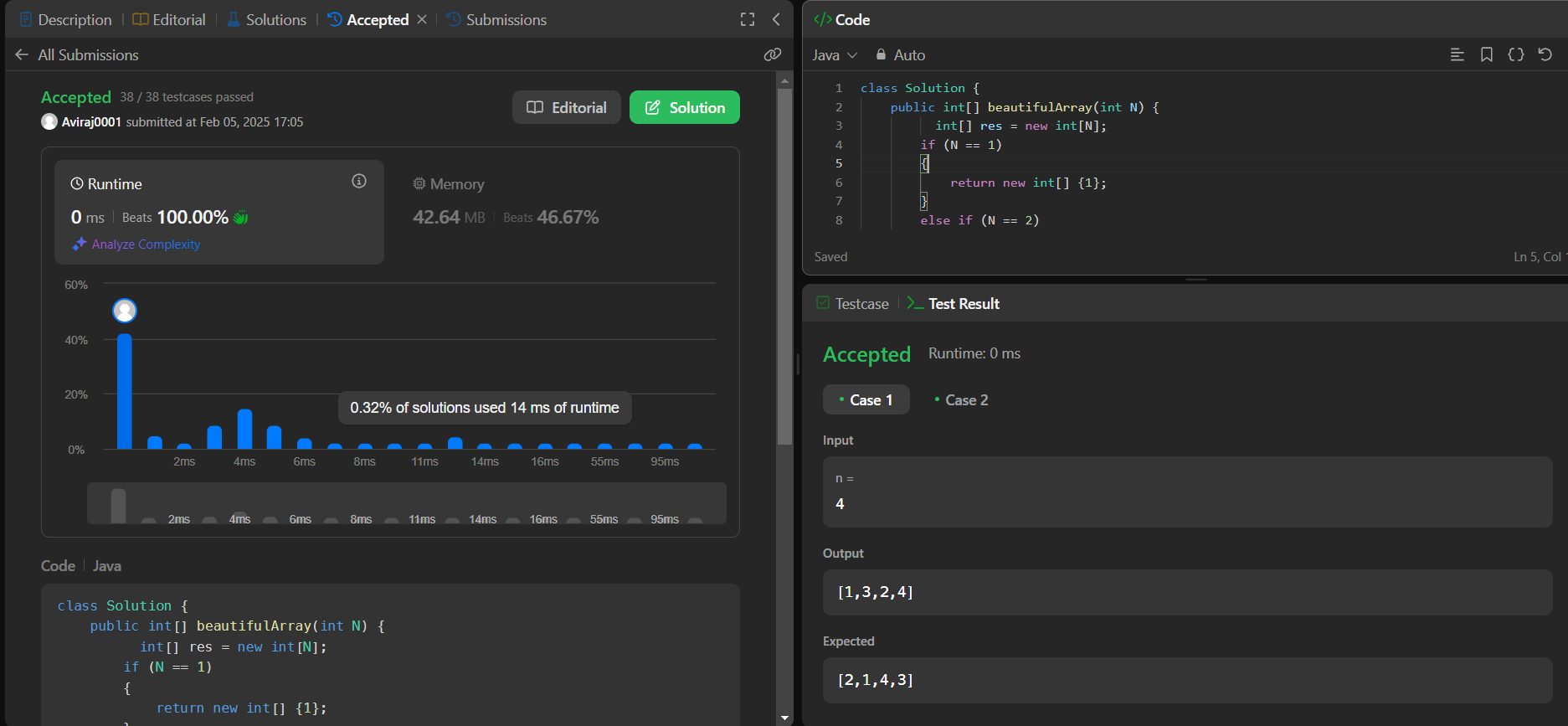
}

return res;

}

}

**Output:**

****

**Q. Reverese Pairs.**

**Code:**

class Solution {

public void merge(int[] arr, int low, int mid, int high) {

ArrayList<Integer> temp = new ArrayList<>();

int left = low;

int right = mid+1;

while(left <= mid && right <= high) {

if(arr[left] <= arr[right]) {

temp.add(arr[left++]);

} else {

temp.add(arr[right++]);

}

}

while(left <= mid) temp.add(arr[left++]);

while(right <= high) temp.add(arr[right++]);

for(int i=low; i<=high; i++) {

arr[i] = temp.get(i-low);

}

}

public int countPairs(int[] arr, int low, int mid, int high) {

int right = mid + 1;

int cnt = 0;

for(int i=low; i<=mid; i++) {

while(right <= high && (long) arr[i] > 2L \* arr[right])

right++;

cnt += (right - (mid + 1));

}

return cnt;

}

public int mergeSort(int[] arr, int low, int high) {

int cnt = 0;

if(low >= high) return cnt;

int mid = (low + high) / 2;

cnt += mergeSort(arr,low,mid);

cnt += mergeSort(arr,mid+1,high);

cnt += countPairs(arr,low,mid,high);

merge(arr,low,mid,high);

return cnt;

}

public int reversePairs(int[] nums) {

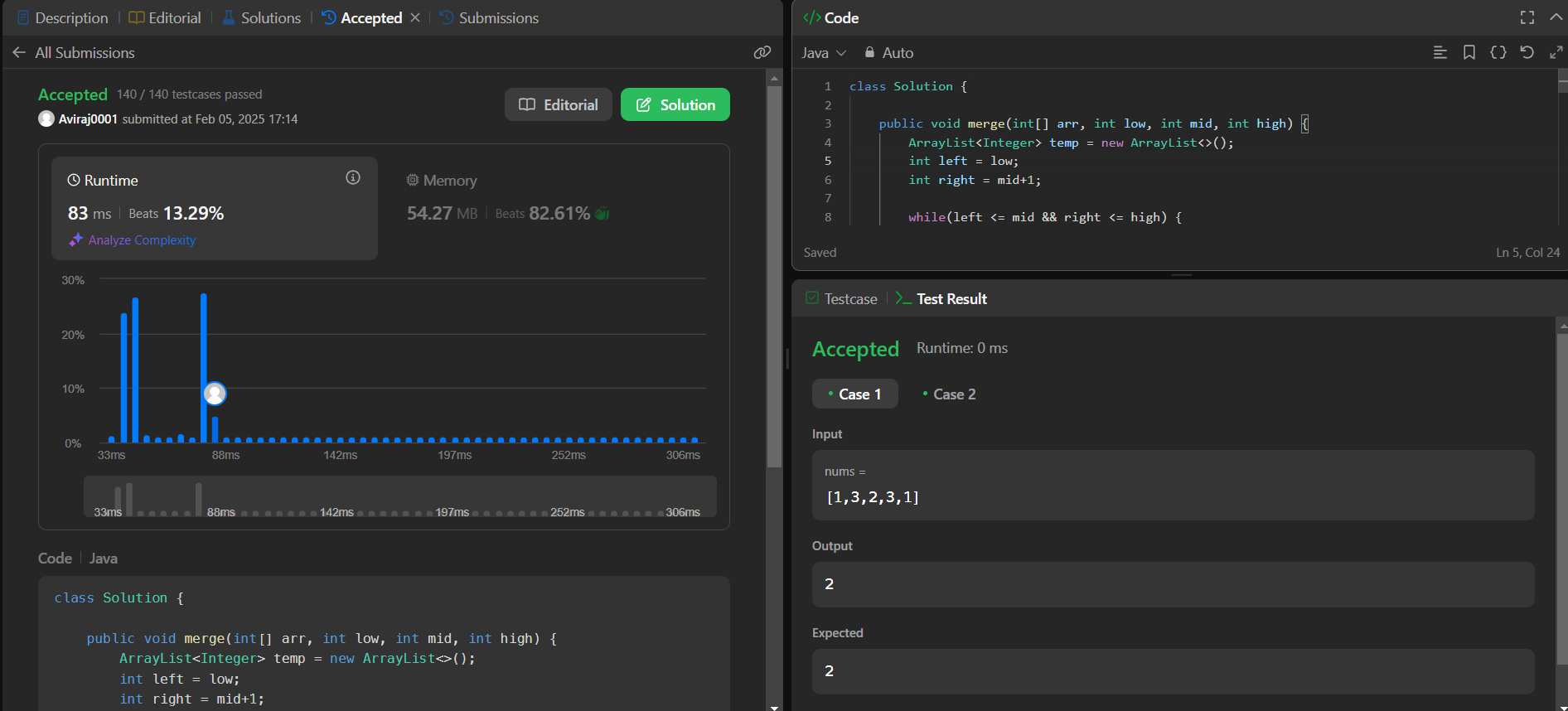
int n = nums.length;

return mergeSort(nums, 0, n-1);

}

}

**Output:**



**Q. Longest Increasing Subsequence II**

**Code:**

class Solution {

public int lengthOfLIS(int[] nums, int k) {

var minVal = Integer.MAX\_VALUE;

var maxVal = Integer.MIN\_VALUE;

for (int n : nums) {

minVal = Math.min(minVal, n);

maxVal = Math.max(maxVal, n);

}

// build the segment tree

var dp = new SegmentTree(minVal, maxVal);

for (int num : nums) {

// find longest chain in range

var preMax = 1 + dp.rangeMaxQuery(num - k, num - 1);

// store the results

dp.update(num, preMax);

}

return dp.val;

}

static final class SegmentTree {

SegmentTree left;

SegmentTree right;

final int lo;

final int hi;

int val;

SegmentTree(int lo, int hi) {

this.lo = lo;

this.hi = hi;

if (lo != hi) {

var mid = lo + (hi - lo) / 2;

this.left = new SegmentTree(lo, mid);

this.right = new SegmentTree(mid + 1, hi);

}

}

void update(int index, int val) {

if (index < this.lo || this.hi < index) // out of range

return;

if (lo == hi) { // found node

this.val = val;

return;

}

this.left.update(index, val);

this.right.update(index, val);

this.val = Math.max(this.left.val, this.right.val);

}

int rangeMaxQuery(int lo, int hi) {

if (hi < this.lo || this.hi < lo) // not overlap

return 0;

if (lo <= this.lo && this.hi <= hi) // in range

return this.val;

return Math.max(this.left.rangeMaxQuery(lo, hi), this.right.rangeMaxQuery(lo, hi));

}

@Override

public String toString() {

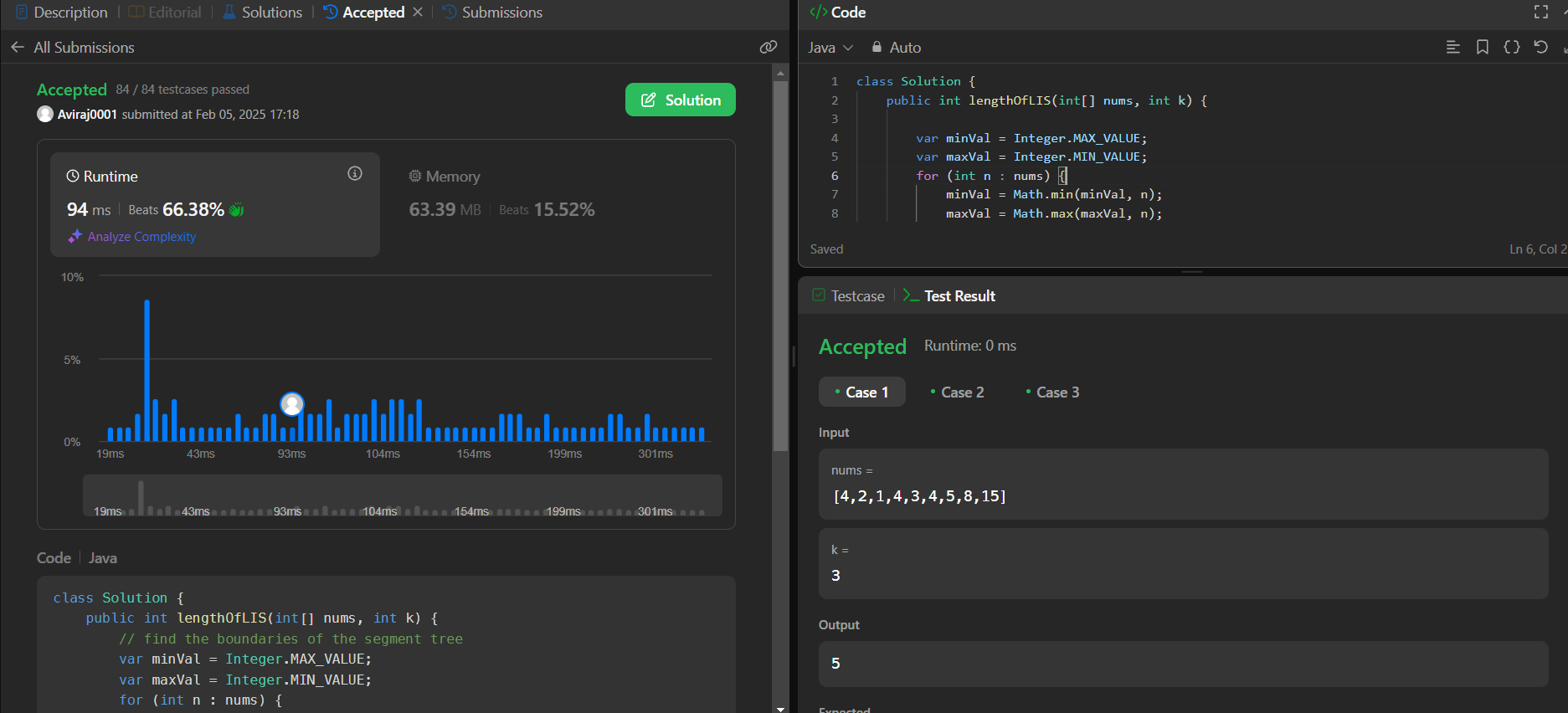
return "[" + lo + "," + hi + "]->" + val;

}

}

}

Output:



**Q. First Bad Version.**

**Code:**

public class Solution extends VersionControl {

public int firstBadVersion(int n) {

int f = 1, l = n;

while(f<=l){

int m = f+(l-f)/2;

if(!isBadVersion(m)){

f = m+1;

}else{

l = m-1;

}

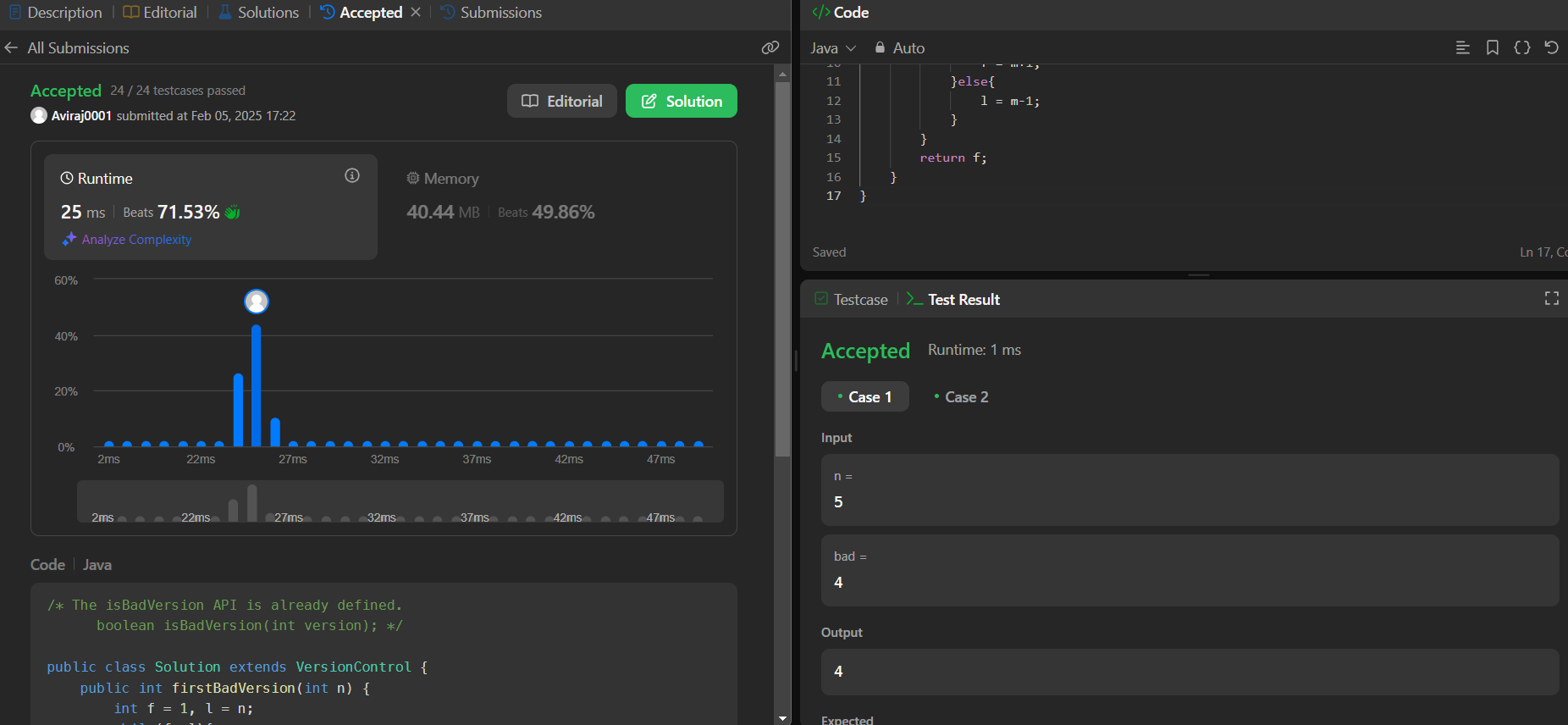
}

return f;

}

}

**Output:**

****

**Q. Sort Colors.**

**Code:**

class Solution {

public void sortColors(int[] nums) {

int low = 0, mid = 0, high = nums.length - 1;

while (mid <= high) {

if (nums[mid] == 0) {

swap(nums, low, mid);

low++;

mid++;

} else if (nums[mid] == 1) {

mid++;

} else {

swap(nums, mid, high);

high--;

}

}

}

private void swap(int[] nums, int i, int j) {

int temp = nums[i];

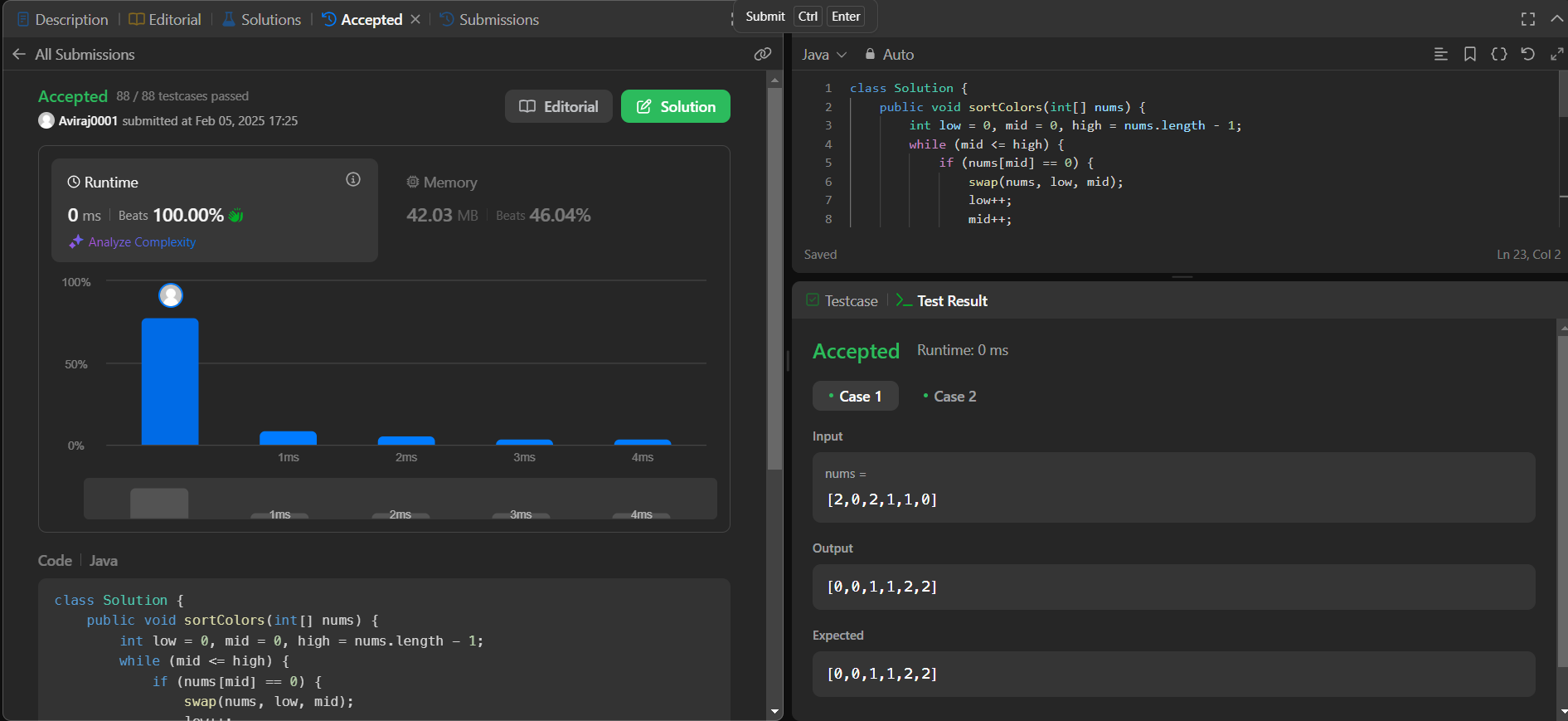
nums[i] = nums[j];

nums[j] = temp;

}

}

**Output:**

****

**Q. Top K Frequent Elements**

**Code:**

class Solution {

public int[] topKFrequent(int[] nums, int k) {

List<Integer>[] bucket = new List[nums.length + 1];

HashMap<Integer, Integer> hm = new HashMap<>();

for (int num : nums) {

hm.put(num, hm.getOrDefault(num,0) + 1);

}

for (int key : hm.keySet()) {

int freq = hm.get(key);

if (bucket[freq] == null) {

bucket[freq] = new ArrayList<>();

}

bucket[freq].add(key);

}

int[] ans = new int[k];

int pos = 0;

for (int i = bucket.length - 1; i >= 0; i--) {

if (bucket[i] != null) {

for (int j = 0; j < bucket[i].size() && pos < k; j++) {

ans[pos] = bucket[i].get(j);

pos++;

}

}

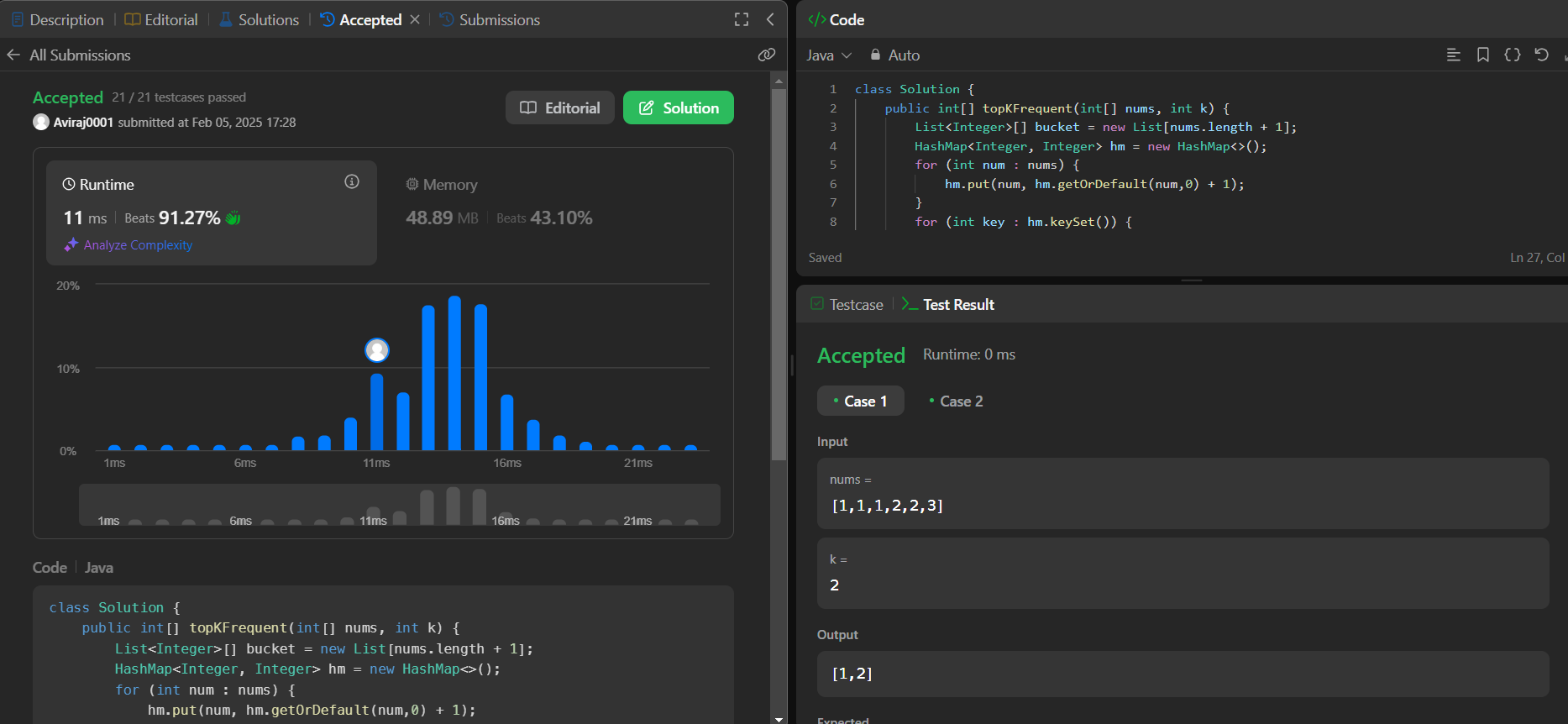
}

return ans;

}

}

**Output:**

****

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

**Code:**

class Solution {

public int findKthLargest(int[] nums, int k) {

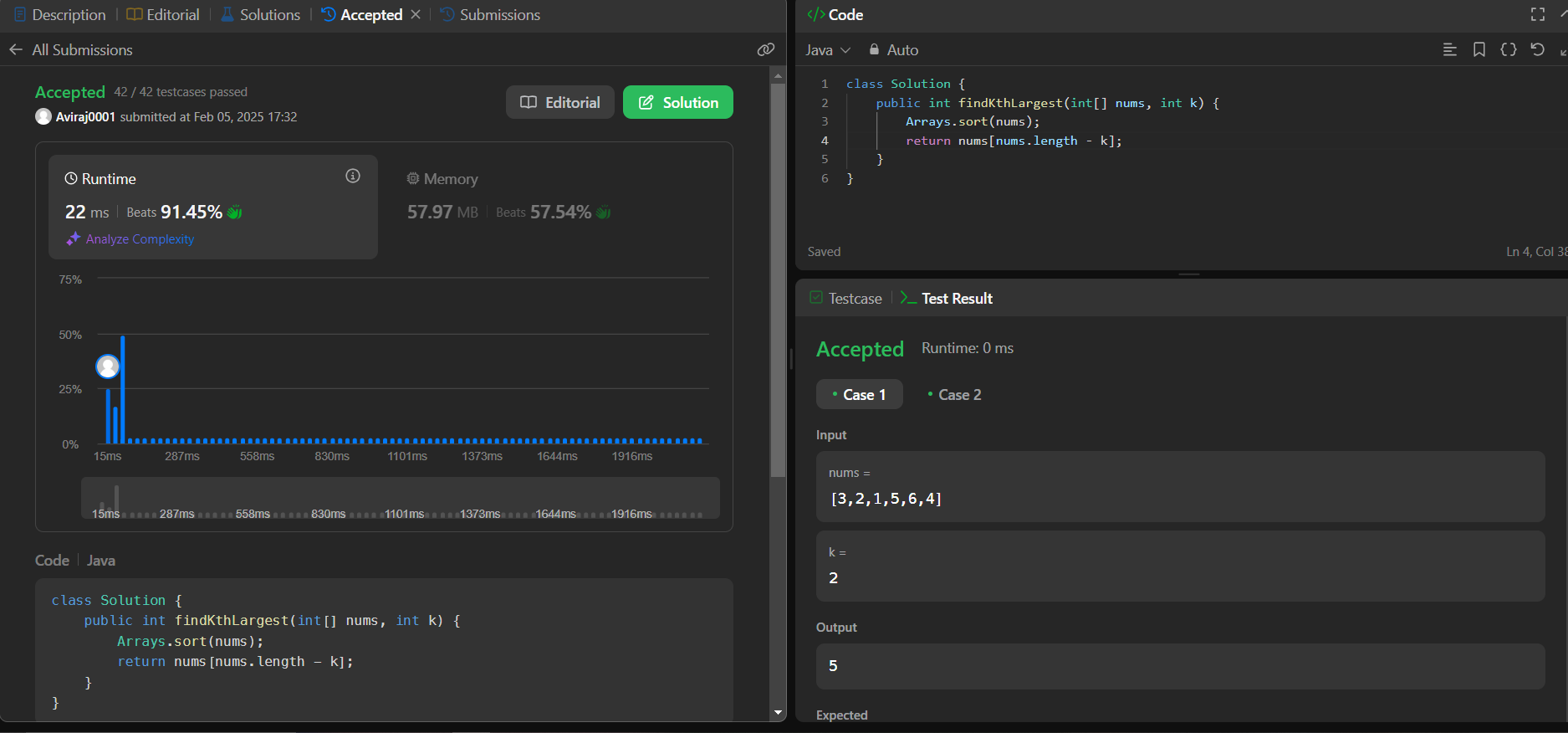
Arrays.sort(nums);

return nums[nums.length - k];

}

}

**Output:**

****

**Q.** **Find Peak Element.**

**Code:**

class Solution {

public int findPeakElement(int[] nums) {

int n = nums.length;

int ans = Integer.MIN\_VALUE, max = Integer.MIN\_VALUE, result = 0;

int left=0, right=n-1;

while(left <= right){

max = Math.max(nums[left], nums[right]);

ans = Math.max(ans, max);

if(nums[left] == ans) result = left;

else if(nums[right] == ans) result = right;

left++;

right--;

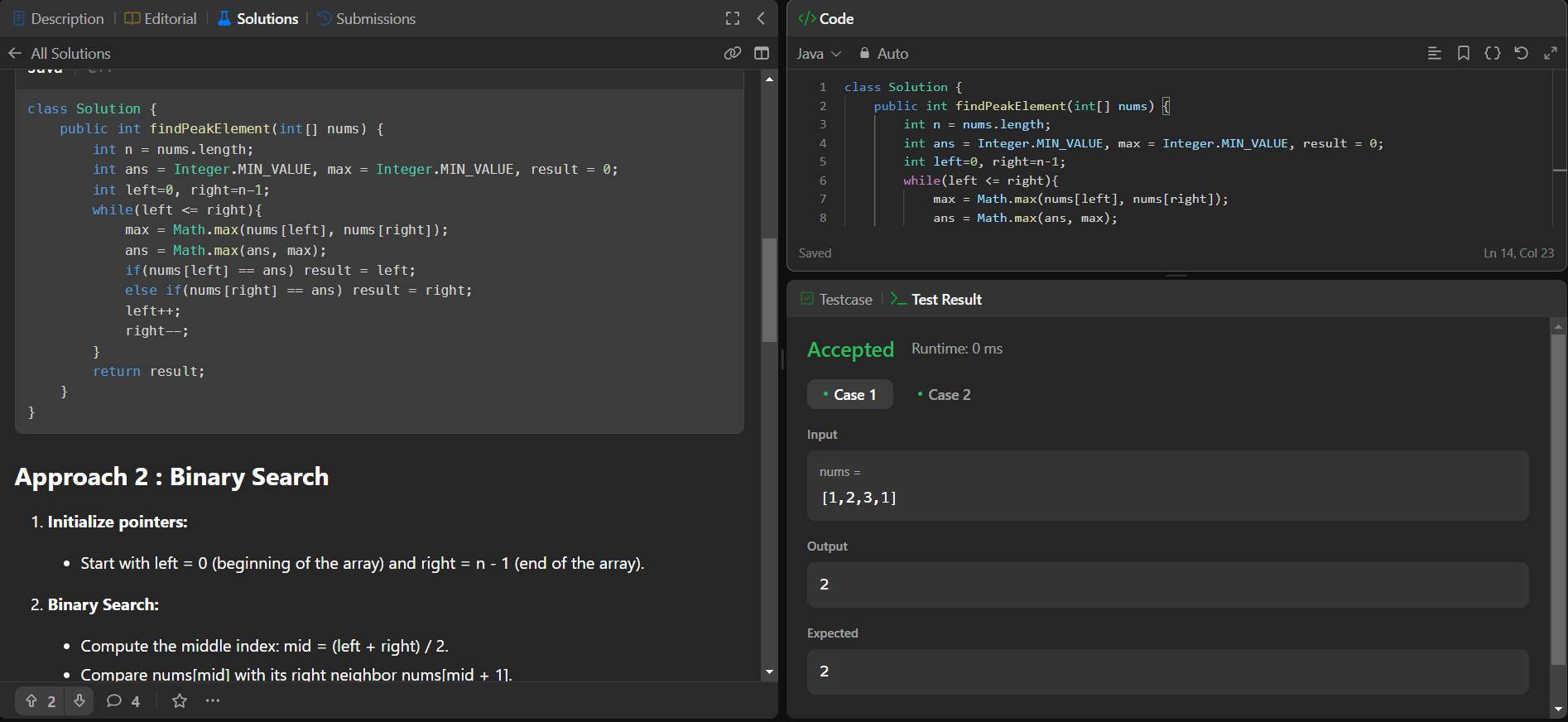
}

return result;

}

}

**Output:**

****

**Q. Merge Intervals.**

**Code:**

import java.util.\*;

class Solution {

public int[][] merge(int[][] intervals) {

Arrays.sort(intervals, (a, b) -> Integer.compare(a[0], b[0]));

int k = 0;

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

if (intervals[k][1] >= intervals[i][0]) {

intervals[k][1] = Math.max(intervals[k][1], intervals[i][1]);

} else {

k++;

intervals[k] = intervals[i];

}

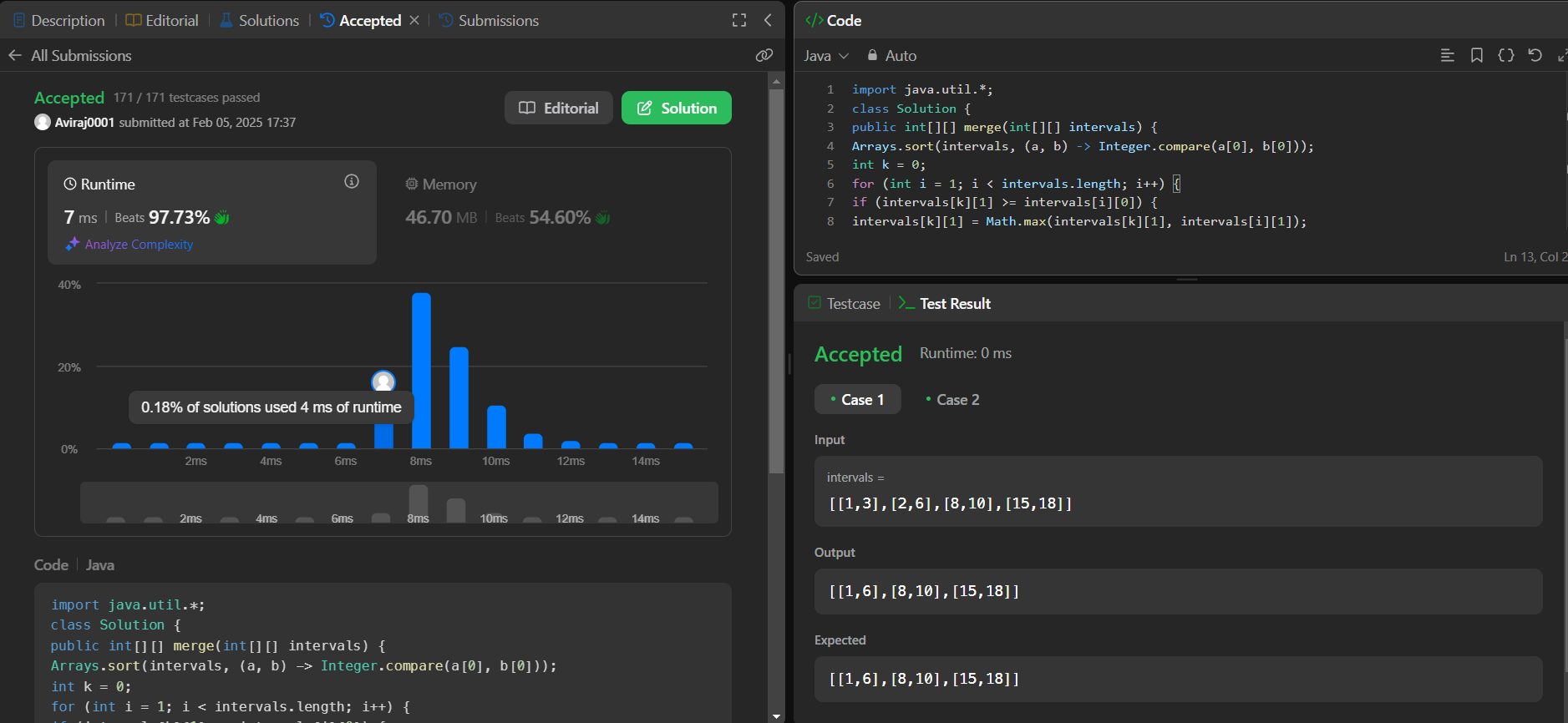
}

return Arrays.copyOfRange(intervals, 0, k + 1);

}

}

**Output:**

****

**Q. Search in Rotated Sorted Array.**

**Code:**

class Solution {

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

int low = 0, high = nums.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

if (nums[mid] == target) {

return mid;

}

if (nums[low] <= nums[mid]) {

if (nums[low] <= target && target < nums[mid]) {

high = mid - 1;

} else {

low = mid + 1;

}

} else {

if (nums[mid] < target && target <= nums[high]) {

low = mid + 1;

} else {

high = mid - 1;

}

}

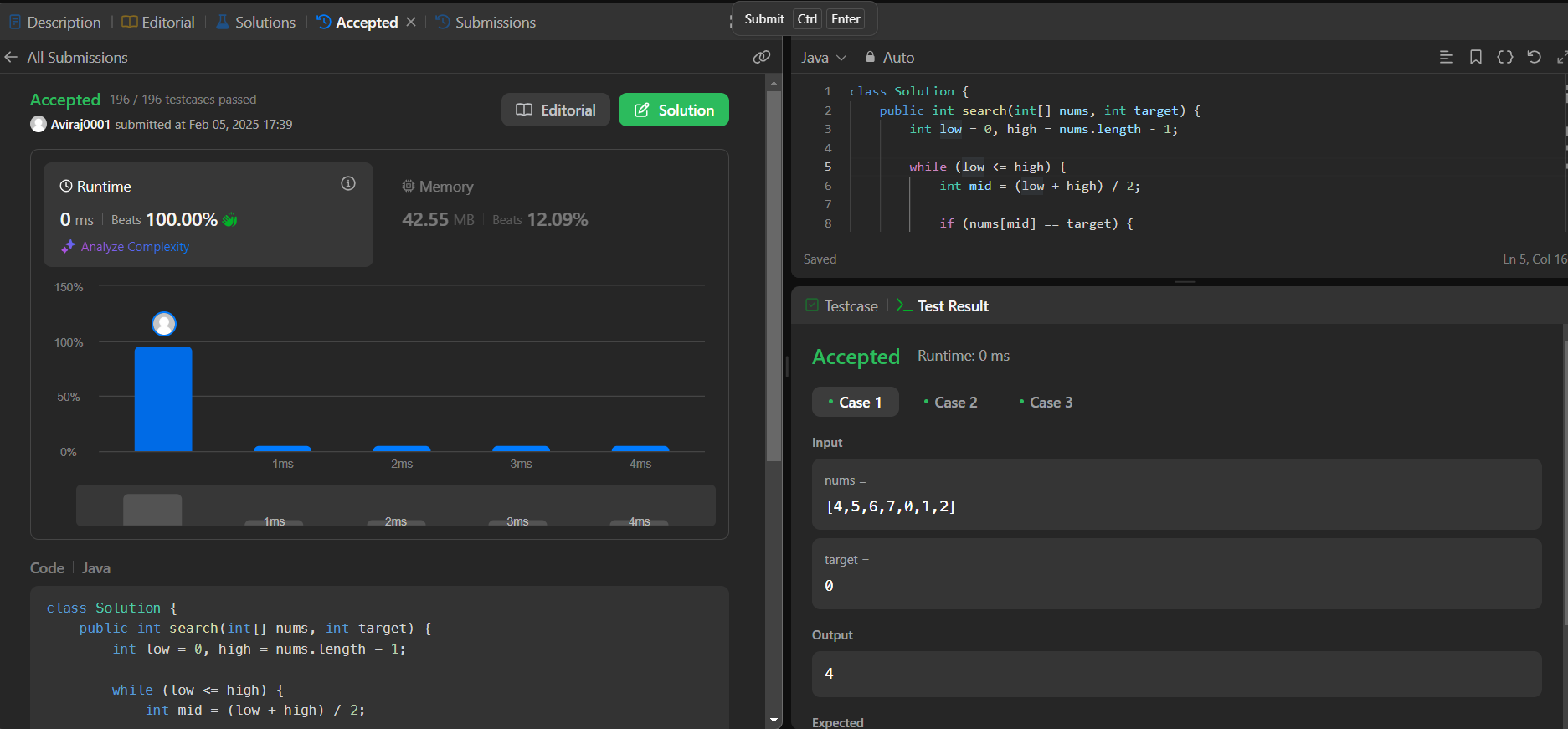
}

return -1;

}

}

**Output:**

****

**Q. Search a 2D Matrix.**

**Code:**

class Solution {

public boolean searchMatrix(int[][] matrix, int target) {

int m = matrix.length;

int n = matrix[0].length;

int row = 0, col = n - 1;

while (row < m && col >= 0) {

if (matrix[row][col] == target) {

return true;

} else if (matrix[row][col] > target) {

col--;

} else {

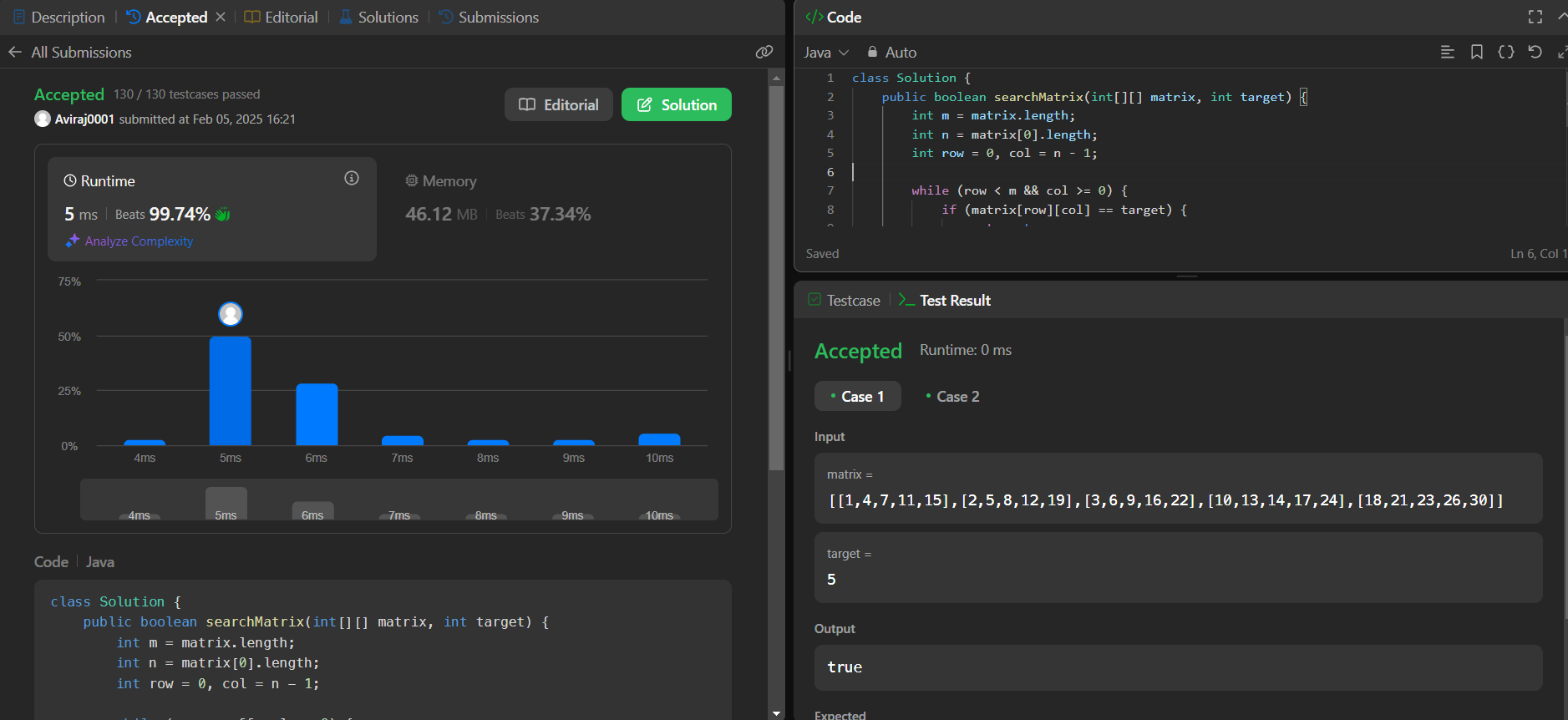
row++;

} }

return false;

}}

**Output:**

****

**Q. Wiggle Sort.**

Code:

class Solution {

public void wiggleSort(int[] nums) {

int n=nums.length-1;

int[] newarr=Arrays.copyOf(nums,nums.length);

Arrays.sort(newarr);

for(int i=1;i<nums.length;i+=2)

nums[i]=newarr[n--];

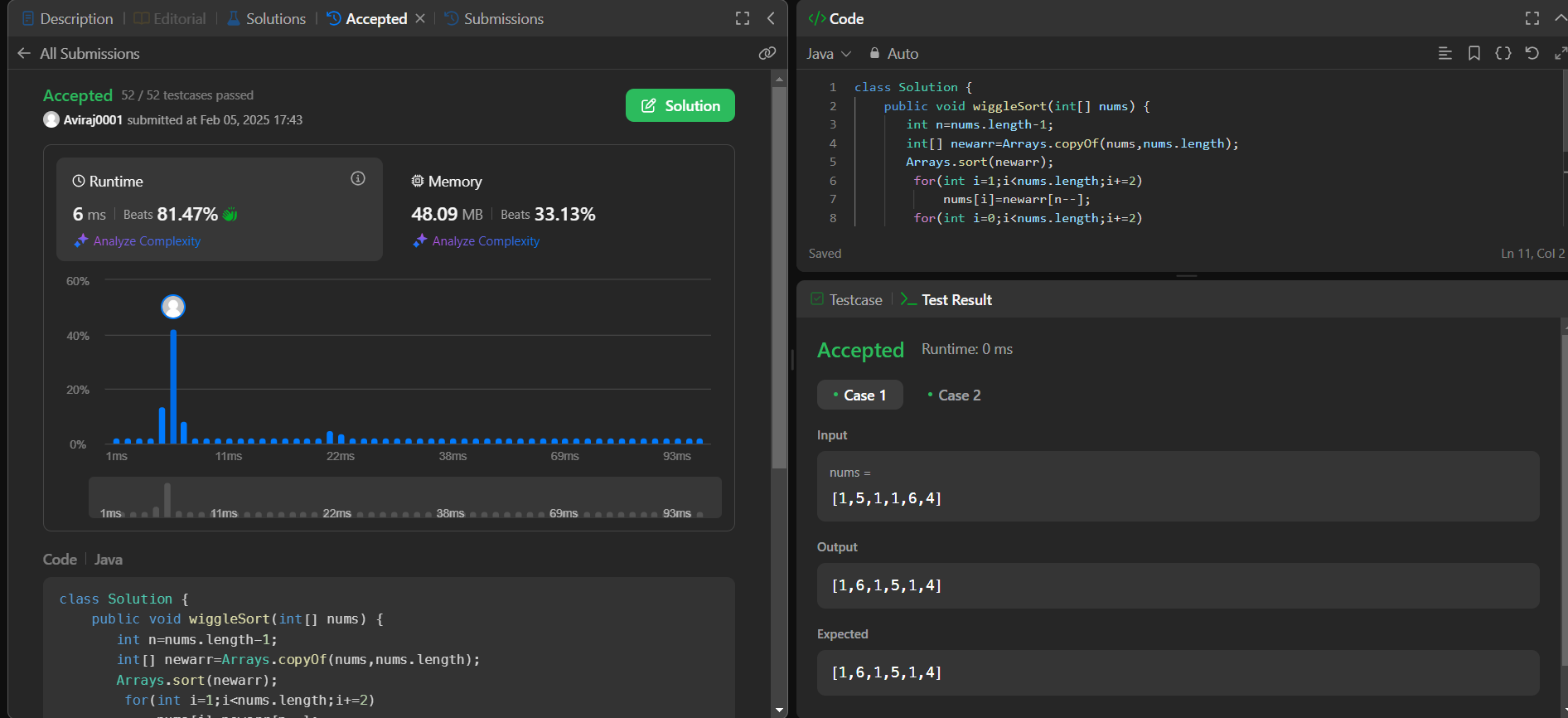
for(int i=0;i<nums.length;i+=2)

nums[i]=newarr[n--];

}

}

**Output:**

****

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

**Code:**

class Solution {

public int kthSmallest(int[][] matrix, int k) {

int n= matrix.length;

int low = matrix[0][0];

int high = matrix[n-1][n-1];

while(low < high){

int mid = low + (high - low)/2;

int count = lessEqual(matrix,mid);

if(count < k){

low = mid+1;

}

else{

high = mid;

}

}

return low;

}

public int lessEqual(int[][] matrix, int target){

int count = 0 , len = matrix.length, i = len-1, j=0;

while(i >=0 && j<len){

if(matrix[i][j] > target){

i--;

}

else

{

count = count + i +1;

j++;

}

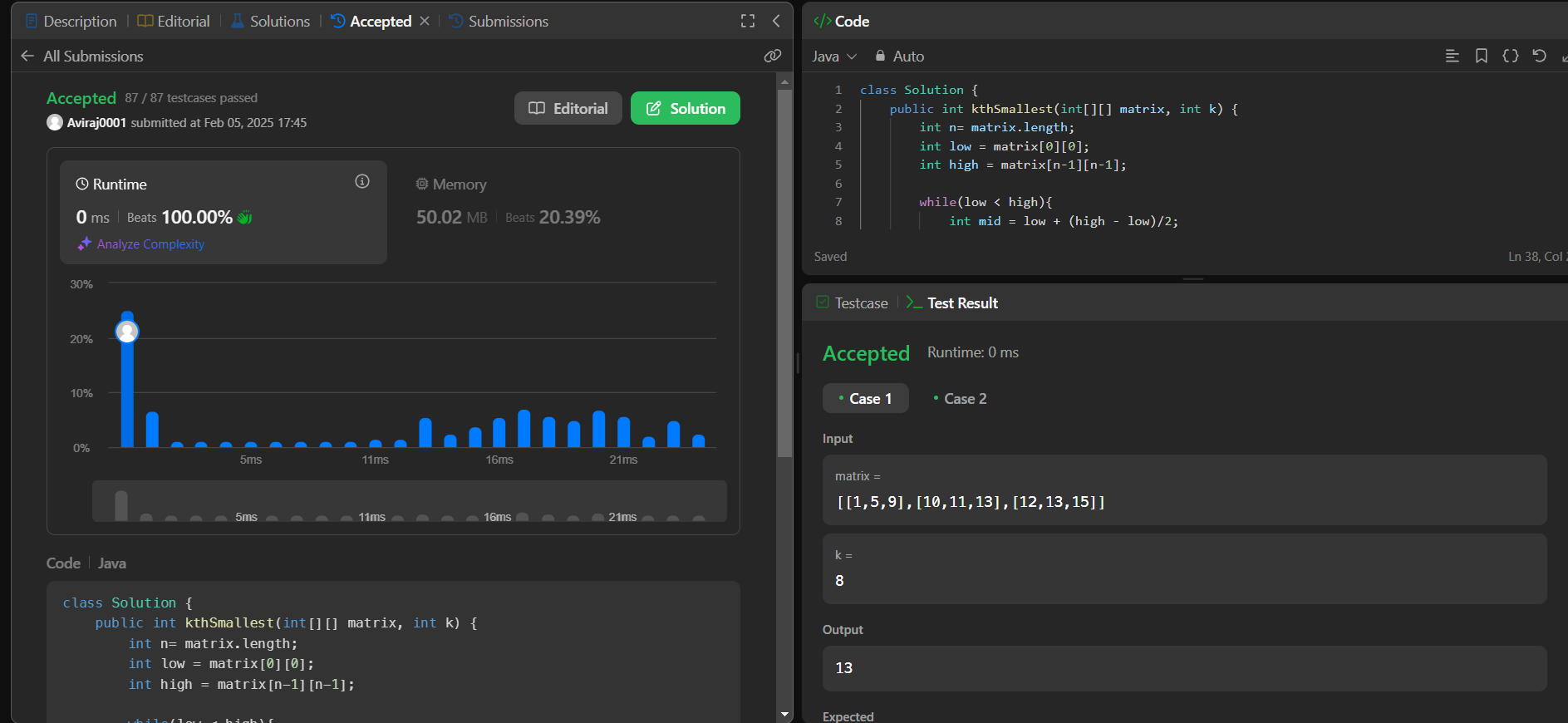
}

return count;

}

}

**Output:**

****

**Q. Median of Two Sorted Arrays.**

**Code:**

public class Solution {

public double findMedianSortedArrays(int[] nums1, int[] nums2) {

if (nums1.length > nums2.length) {

int[] temp = nums1;

nums1 = nums2;

nums2 = temp;

}

int m = nums1.length;

int n = nums2.length;

int low = 0, high = m;

while (low <= high) {

int partitionX = (low + high) / 2;

int partitionY = (m + n + 1) / 2 - partitionX;

int maxX = (partitionX == 0) ? Integer.MIN\_VALUE : nums1[partitionX - 1];

int maxY = (partitionY == 0) ? Integer.MIN\_VALUE : nums2[partitionY - 1];

int minX = (partitionX == m) ? Integer.MAX\_VALUE : nums1[partitionX];

int minY = (partitionY == n) ? Integer.MAX\_VALUE : nums2[partitionY];

if (maxX <= minY && maxY <= minX) {

if ((m + n) % 2 == 0) {

return (Math.max(maxX, maxY) + Math.min(minX, minY)) / 2.0;

} else {

return Math.max(maxX, maxY);

}

} else if (maxX > minY) {

high = partitionX - 1;

} else {

low = partitionX + 1;

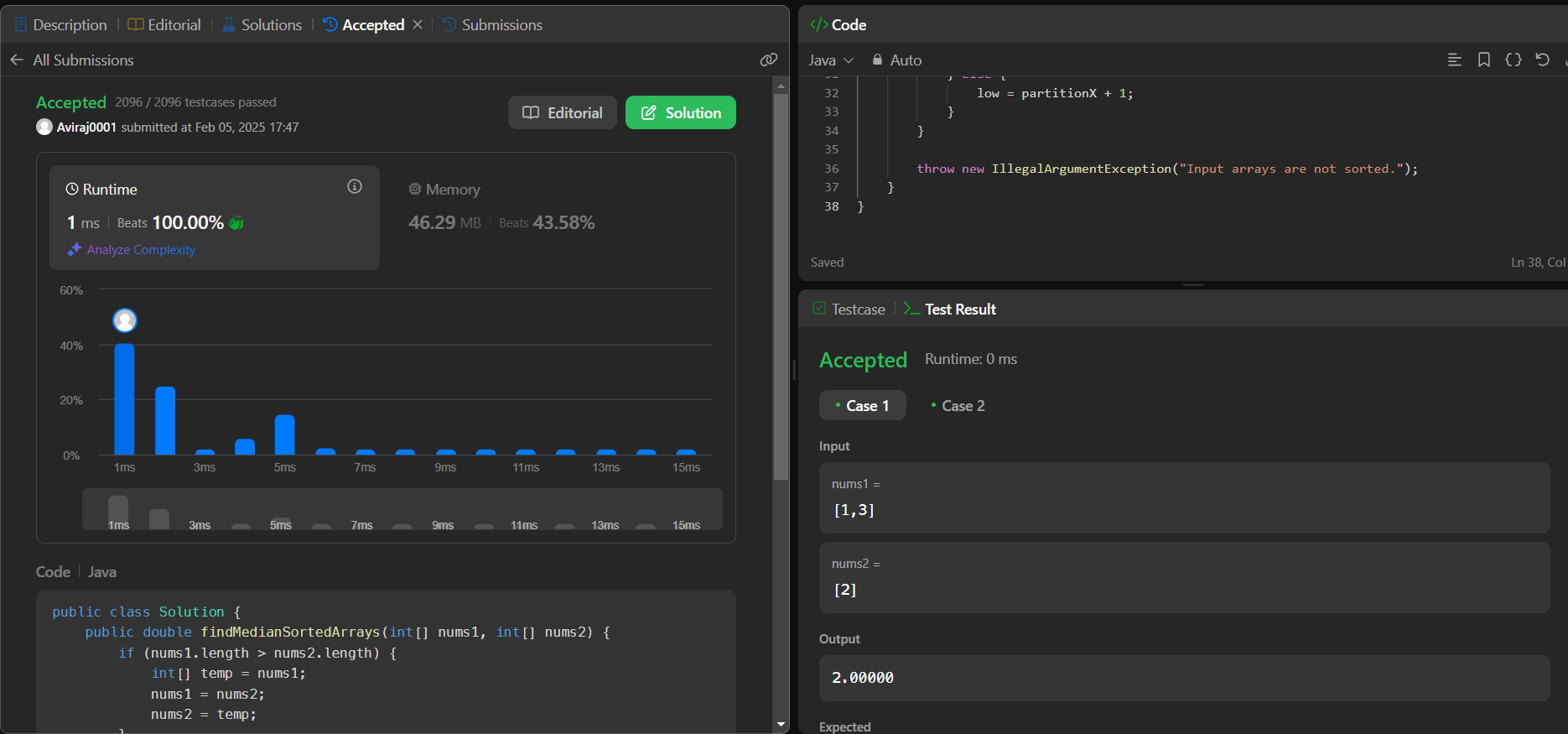
}

}

throw new IllegalArgumentException("Input arrays are not sorted.");

}

}  
**Output:**



**Q. The Skyline Problem.**

**Code:**

class Solution {

class KeyPoint {

public int key;

public int height;

public KeyPoint next = null;

public KeyPoint(int key, int height) {

this.key = key;

this.height = height;

}

}

public List<List<Integer>> getSkyline(int[][] buildings) {

List<List<Integer>> res = new ArrayList<>();

KeyPoint dummy = new KeyPoint(-1, 0); // dummy head

KeyPoint pre = dummy;

for (int[] bd : buildings) {

int L = bd[0];

int R = bd[1];

int H = bd[2];

while (pre.next != null && pre.next.key <= L)

pre = pre.next;

int preH = pre.height;

if (pre.key == L)

pre.height = Math.max(pre.height, H);

else if (pre.height < H) {

KeyPoint next = pre.next;

pre.next = new KeyPoint(L, H);

pre = pre.next;

pre.next = next;

}

KeyPoint preIter = pre;

KeyPoint curIter = pre.next;

while (curIter != null && curIter.key < R) {

preH = curIter.height;

curIter.height = Math.max(curIter.height, H);

if (curIter.height == preIter.height)

preIter.next = curIter.next;

else

preIter = curIter;

curIter = curIter.next;

}

if (preIter.height != preH && preIter.key != R && (curIter == null || curIter.key != R)) {

KeyPoint next = preIter.next;

preIter.next = new KeyPoint(R, preH);

preIter.next.next = next;

}

}

KeyPoint first = dummy;

KeyPoint second = dummy.next;

while (second != null) {

if (second.height != first.height)

res.add(Arrays.asList(second.key, second.height));

first = first.next;

second = second.next;

}

return res;

}

}

**Output:**

