**NAME : KASHISH CHHIMWAL**

**UID : 22BCS14983**

**SECTION : 605(B)**

**Question 1: Longest Nice Substring:**

CODE:

class Solution {

public:

    string longestNiceSubstring(string s) {

         if (s.size() < 2) return ""; // A single letter cannot be nice

    unordered\_set<char> charSet(s.begin(), s.end());

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

        char c = s[i];

        if (charSet.count(tolower(c)) && charSet.count(toupper(c)))

            continue; // This character is nice

        // Split at character 'c' and find longest nice substring

        string left = longestNiceSubstring(s.substr(0, i));

        string right = longestNiceSubstring(s.substr(i + 1));

        return left.size() >= right.size() ? left : right; // Return the longer substring

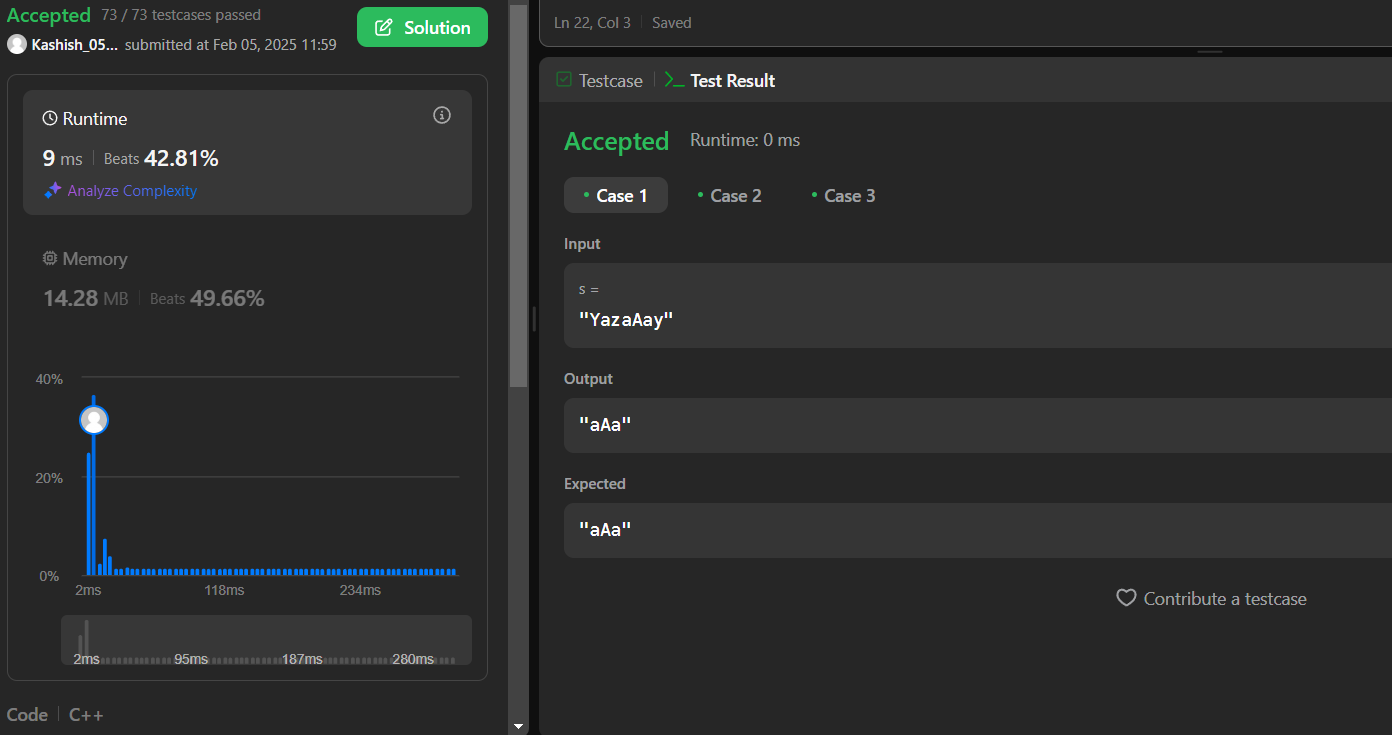
    }

    return s; // If no split happens, entire string is nice}

    }

};

OUTPUT:

s the signed integer -107374182

**Question 2 : REVERSE BITS**

**CODE:**

class Solution {

public:

    uint32\_t reverseBits(uint32\_t n) {

         uint32\_t result = 0;

    for (int i = 0; i < 32; i++) {

        result = (result << 1) | (n & 1);M

        n >>= 1;

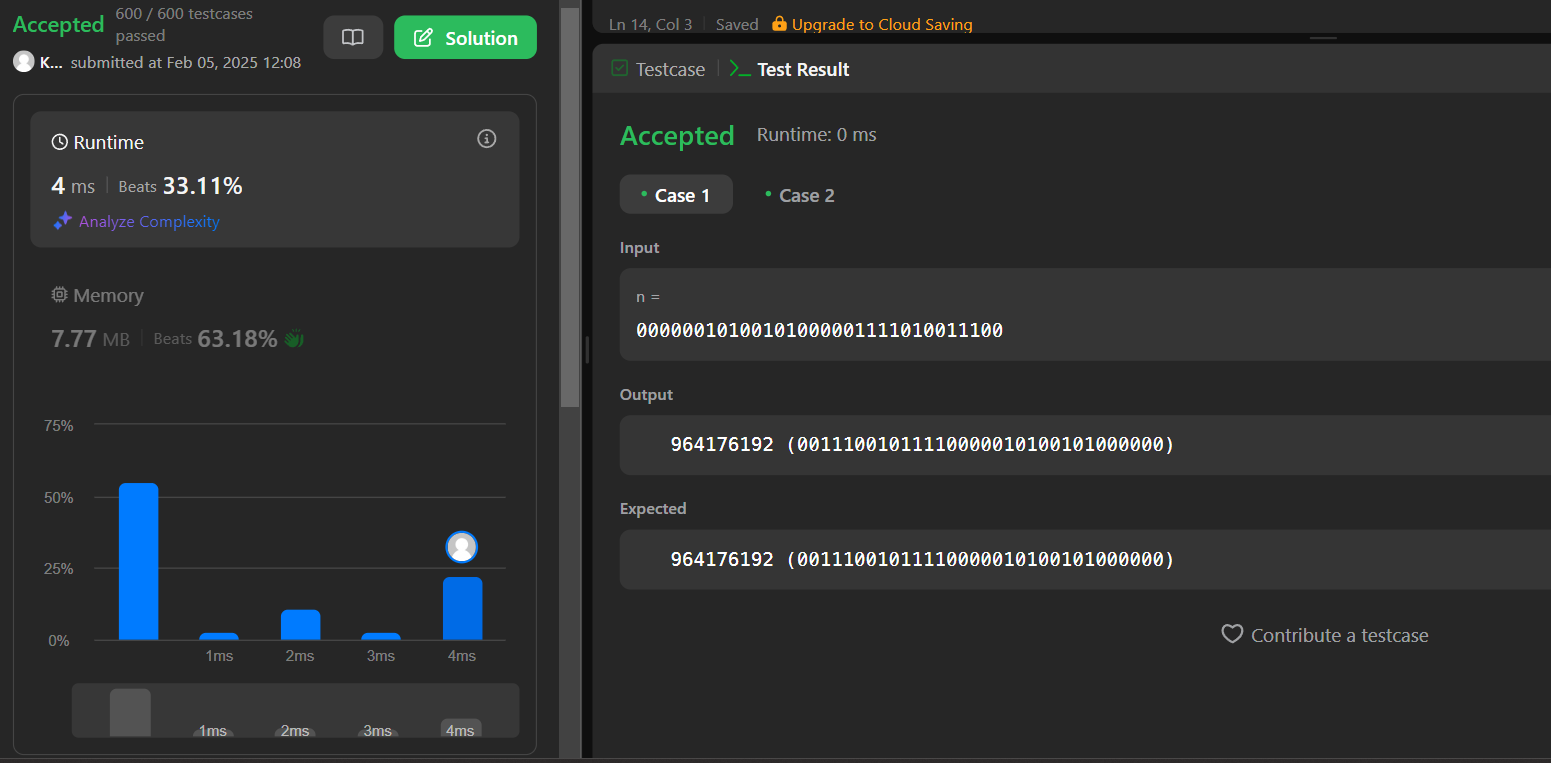
    }

    return result;

    }

};

**OUTPUT:**



**Question 3: NUMBER OF 1 BIT:**

CODE:

class Solution {

public:

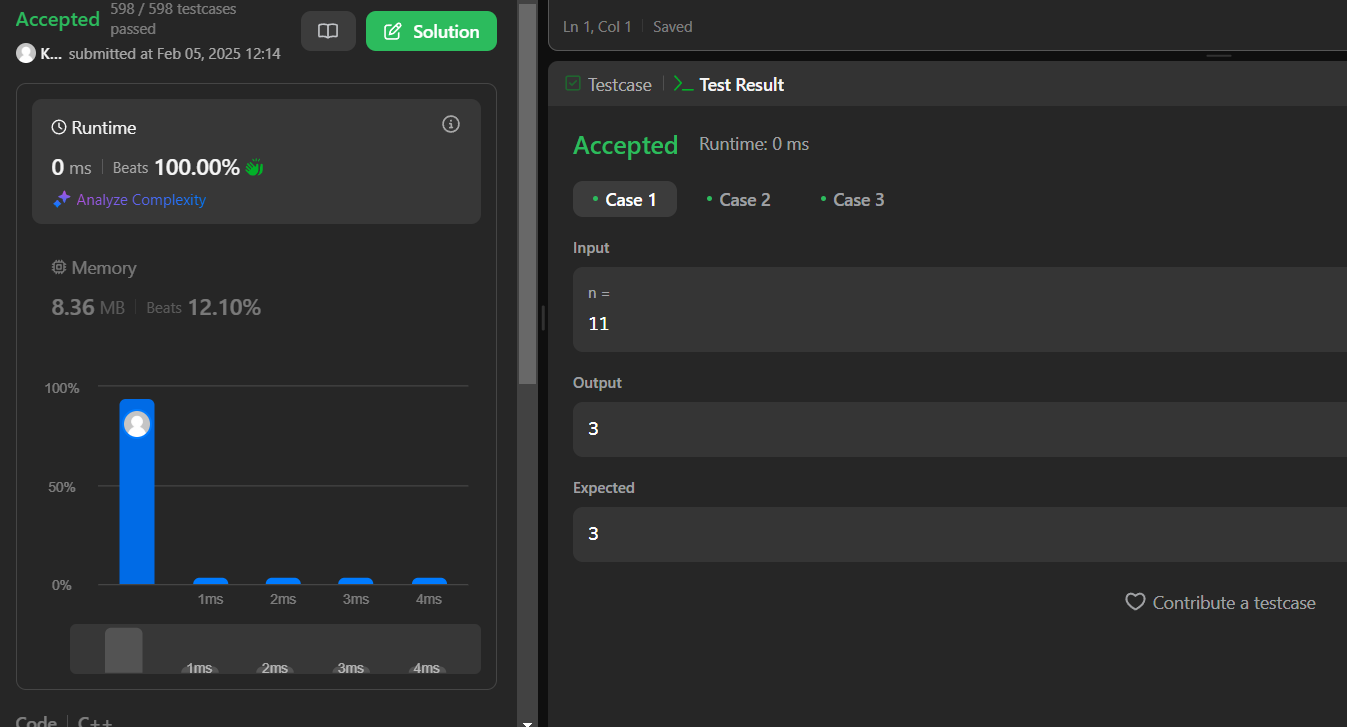
    int hammingWeight(int n) {

        return \_\_builtin\_popcount(n);

    }

};

OUTPUT:



Question 4: MAXIMUM SUBARRAY

CODE:

class Solution {

public:

    int maxSubArray(vector<int>& nums) {

        int maxSum = INT\_MIN;  // Initialize to the smallest integer

        int currentSum = 0;  // Sum of the subarray currently being considered

        for (int num : nums) {

            currentSum += num;  // Add the current element to the sum

            maxSum = max(maxSum, currentSum);  // Update the max sum if needed

            if (currentSum < 0) {

                currentSum = 0;  // If the current sum is negative, reset it to 0

            }

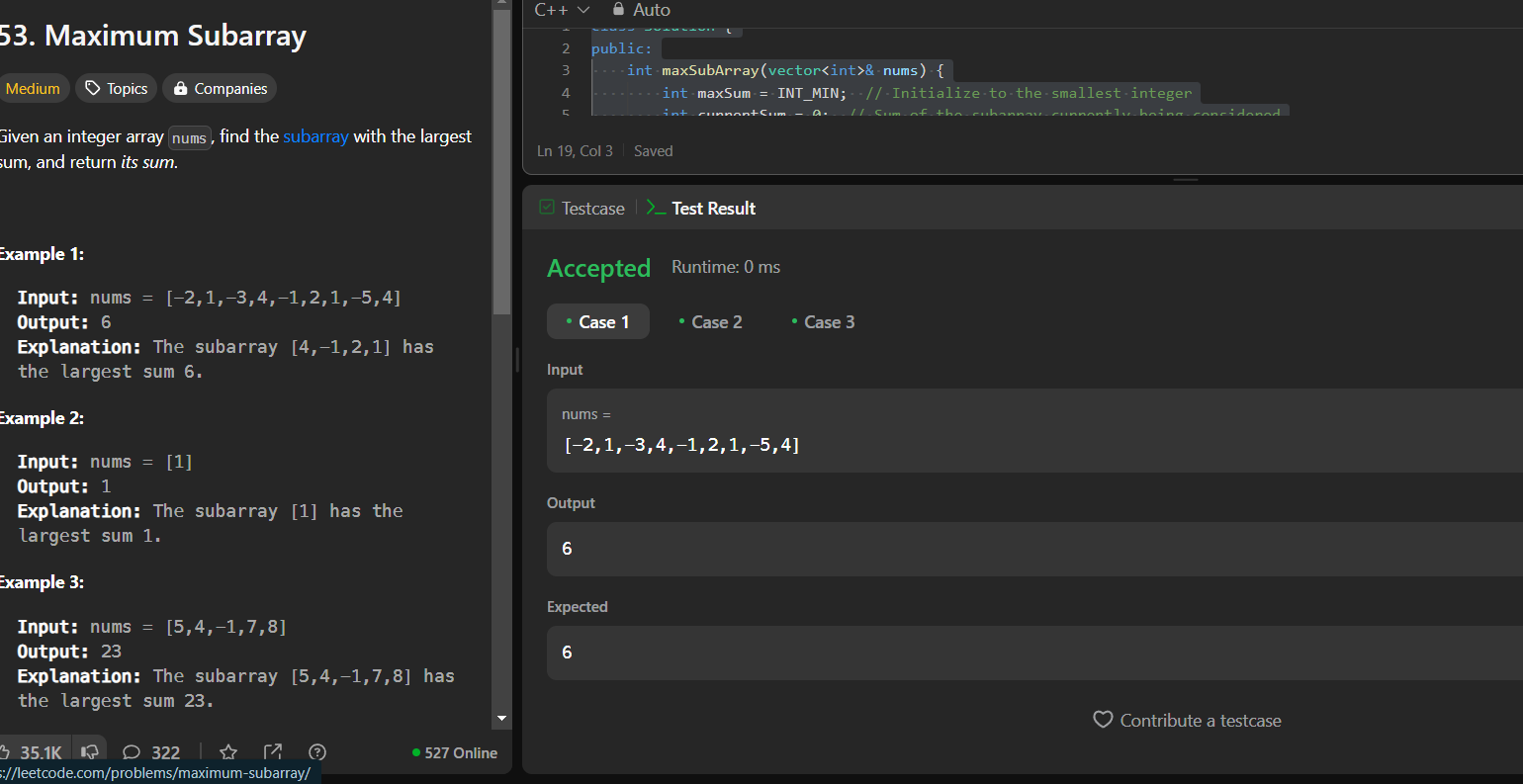
        }

        return maxSum;  // Return the maximum sum found

    }

};

Output:



Question 5: Search a 2D Matrix:

CODE:

class Solution {

public:

    bool searchMatrix(vector<vector<int>>& matrix, int target) {

        int m = matrix.size();

        int n = matrix[0].size();

        int row = 0, col = n - 1;  // Start at the top-right corner

        while (row < m && col >= 0) {

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

                return true;  // Target found

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

                col--;  // Move left

            } else {

                row++;  // Move down

            }

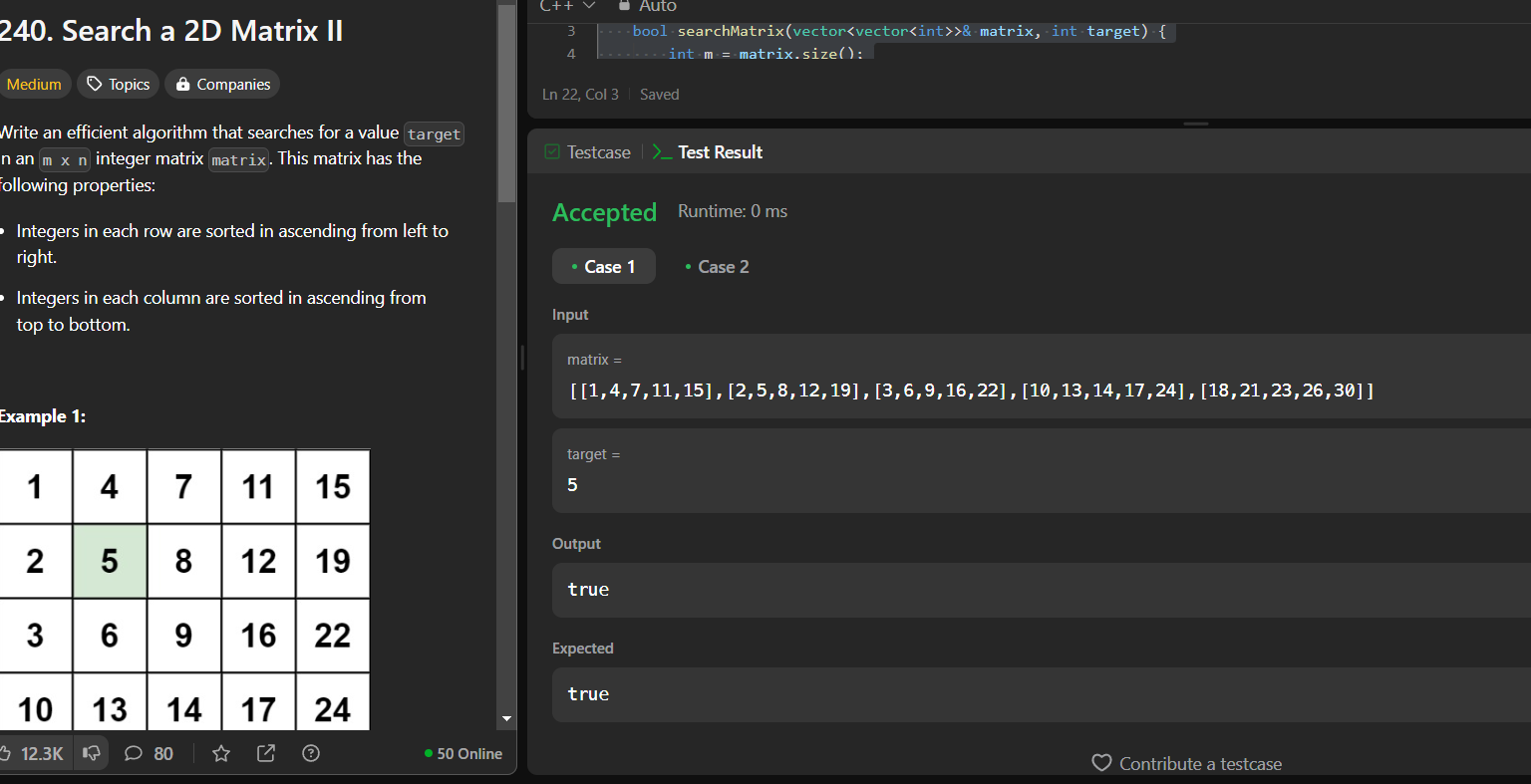
        }

        return false;  // Target not found

    }

};

OUTPUT:



Question 6: MEDIAN OF TWO SORTED ARRAYS:

Code:

class Solution {

public:

    double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {

        if (nums1.size() > nums2.size()) {

            return findMedianSortedArrays(nums2, nums1);

        }

        int m = nums1.size();

        int n = nums2.size();

        int left = 0, right = m;

        while (left <= right) {

            int partition1 = left + (right - left) / 2;

            int partition2 = (m + n + 1) / 2 - partition1;

            // Edge cases for partitioning the arrays

            int maxLeft1 = (partition1 == 0) ? INT\_MIN : nums1[partition1 - 1];

            int minRight1 = (partition1 == m) ? INT\_MAX : nums1[partition1];

            int maxLeft2 = (partition2 == 0) ? INT\_MIN : nums2[partition2 - 1];

            int minRight2 = (partition2 == n) ? INT\_MAX : nums2[partition2];

            // Check if we found the correct partition

            if (maxLeft1 <= minRight2 && maxLeft2 <= minRight1) {

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

                    // If even, median is the average of the max of left and min of right

                    return (max(maxLeft1, maxLeft2) + min(minRight1, minRight2)) / 2.0;

                } else {

                    // If odd, median is the max of the left partition

                    return max(maxLeft1, maxLeft2);

                }

            } else if (maxLeft1 > minRight2) {

                // Move the partition in nums1 to the left

                right = partition1 - 1;

            } else {

                // Move the partition in nums1 to the right

                left = partition1 + 1;

            }

        }

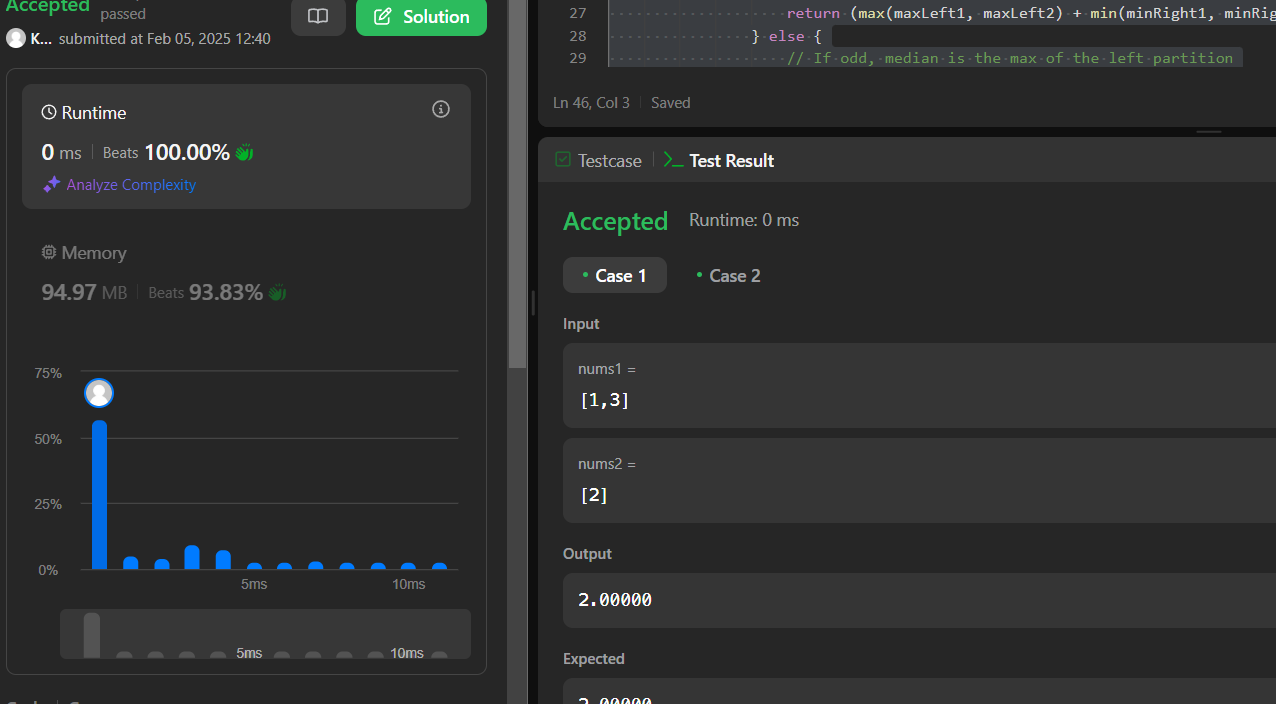
        // If we reach here, the arrays were not sorted properly or invalid input

        throw invalid\_argument("Input arrays are not sorted.");

    }

};

OUTPUT:



Question 7: Kth Smallest element in a sorted Matrix:

CODE: class Solution {

public:

    int kthSmallest(vector<vector<int>>& matrix, int k) {

        int n = matrix.size();

        // Min-Heap to store the elements with their row and column indices

        auto cmp = [](const tuple<int, int, int>& a, const tuple<int, int, int>& b) {

            return get<0>(a) > get<0>(b);  // Compare the actual values in the tuple

        };

        priority\_queue<tuple<int, int, int>, vector<tuple<int, int, int>>, decltype(cmp)> minHeap;

        // Push the first element of each row into the heap

        for (int i = 0; i < n; i++) {

            minHeap.push({matrix[i][0], i, 0});

        }

        // Pop the smallest element and push the next element from the same row

        for (int i = 1; i <= k; i++) {

            auto [val, row, col] = minHeap.top();  // Extract the top element from the heap

            minHeap.pop();

            if (col + 1 < n) {

                minHeap.push({matrix[row][col + 1], row, col + 1});  // Push the next element in the row

            }

            if (i == k) {

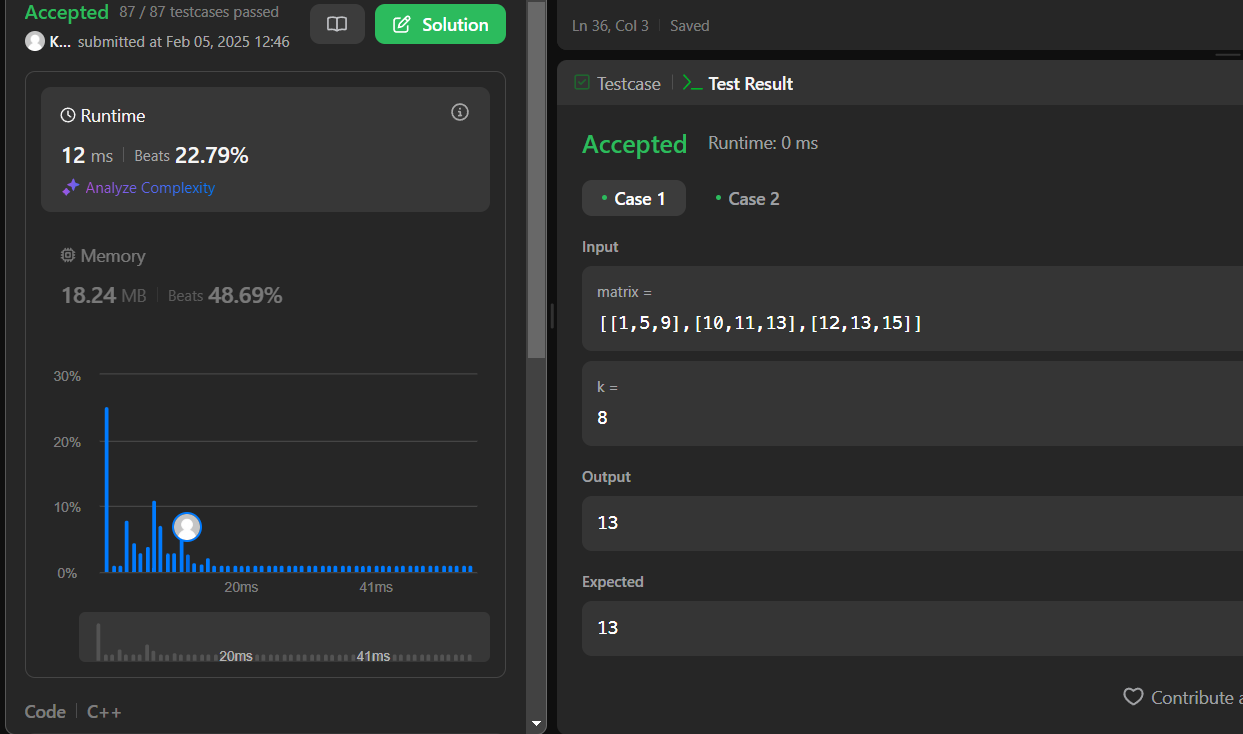
                return val;  // The kth smallest element

            }

        }

        return -1;  // This line should never be reached

    }



**Question 8: WIGGLE SORT:**

CODE:

class Solution {

public:

    void wiggleSort(vector<int>& nums) {

        for (int i = 0; i < nums.size() - 1; i++) {

            // If we are at an even index, we want nums[i] < nums[i + 1]

            if (i % 2 == 0) {

                if (nums[i] > nums[i + 1]) {

                    swap(nums[i], nums[i + 1]);

                }

            }

            // If we are at an odd index, we want nums[i] > nums[i + 1]

            else {

                if (nums[i] < nums[i + 1]) {

                    swap(nums[i], nums[i + 1]);

                }

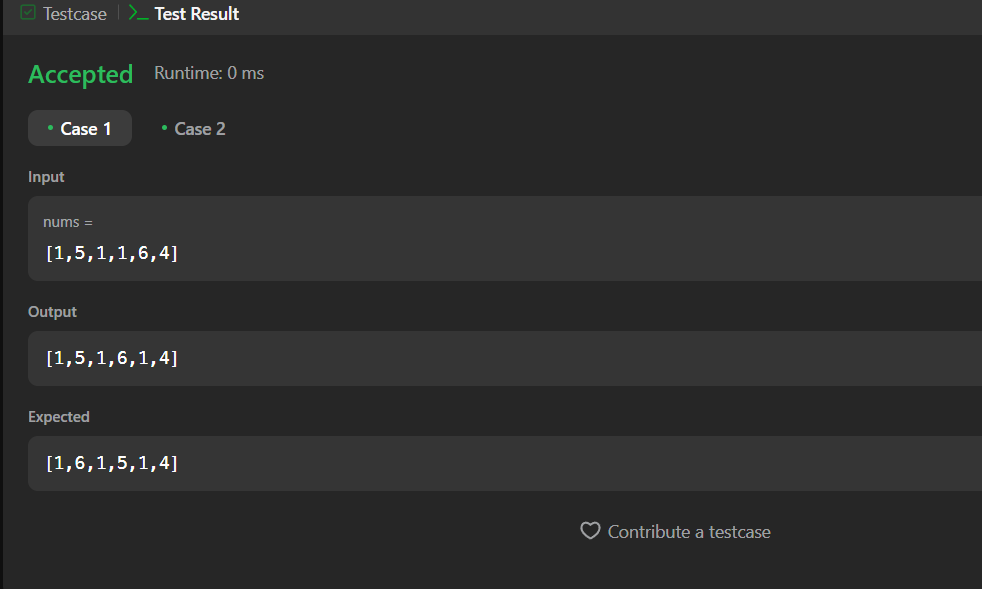
            }

        }

    }

};

OUTPUT:



**Question 9: Search in a 2D matrix:**

CODE:

lass Solution {

public:

    bool searchMatrix(vector<vector<int>>& matrix, int target) {

        int m = matrix.size();

        int n = matrix[0].size();

        int row = 0, col = n - 1;  // Start at the top-right corner

        while (row < m && col >= 0) {

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

                return true;  // Target found

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

                col--;  // Move left

            } else {

                row++;  // Move down

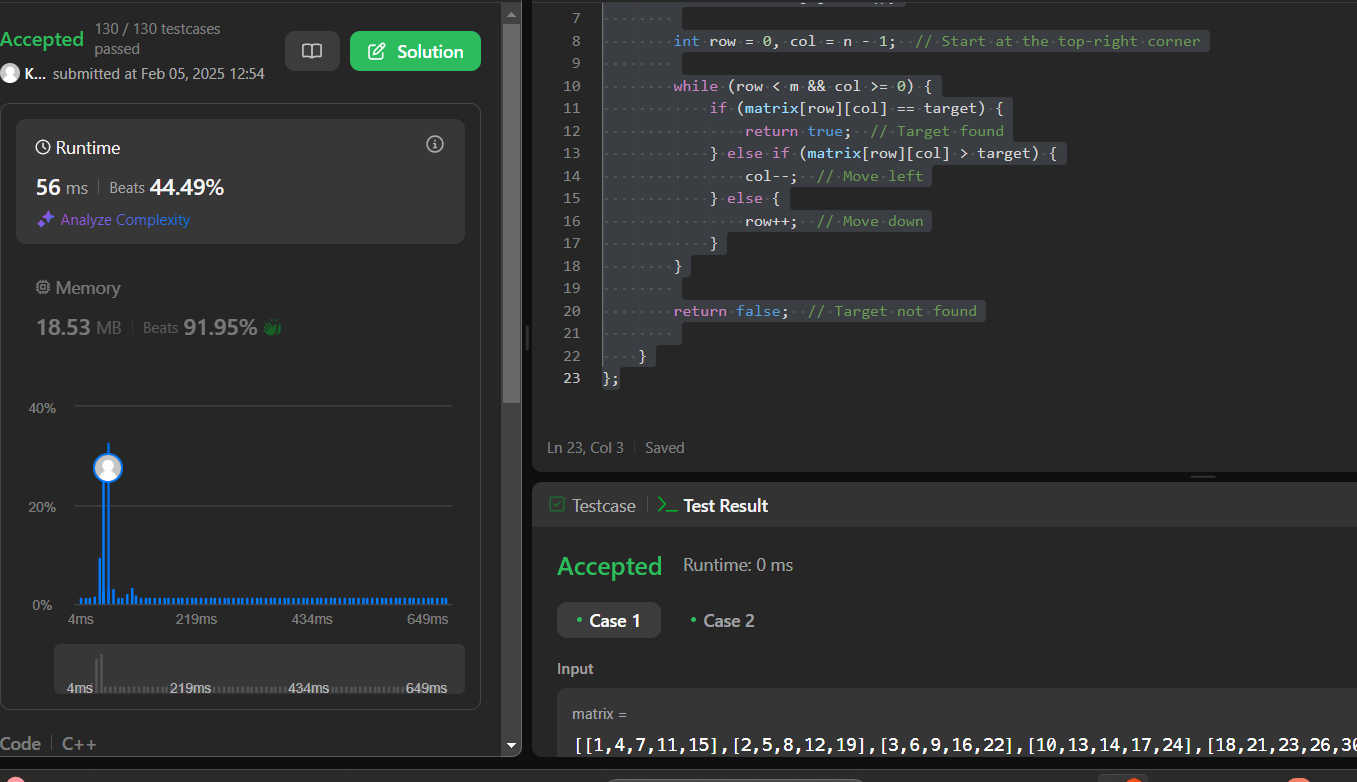
            }

        }

        return false;  // Target not found

    }

};

Output: 

**Question 10: MERGE INTERVALS:**

Code: class Solution {

public:

    vector<vector<int>> merge(vector<vector<int>>& intervals) {

        if (intervals.empty()) return {};

        // Sort intervals based on the start time

        sort(intervals.begin(), intervals.end());

        vector<vector<int>> merged;

        // Start with the first interval

        merged.push\_back(intervals[0]);

        // Iterate through the sorted intervals

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

            // If the current interval overlaps with the last merged interval

            if (merged.back()[1] >= intervals[i][0]) {

                // Merge the intervals by updating the end of the last merged interval

                merged.back()[1] = max(merged.back()[1], intervals[i][1]);

            } else {

                // If no overlap, add the current interval to the result

                merged.push\_back(intervals[i]);

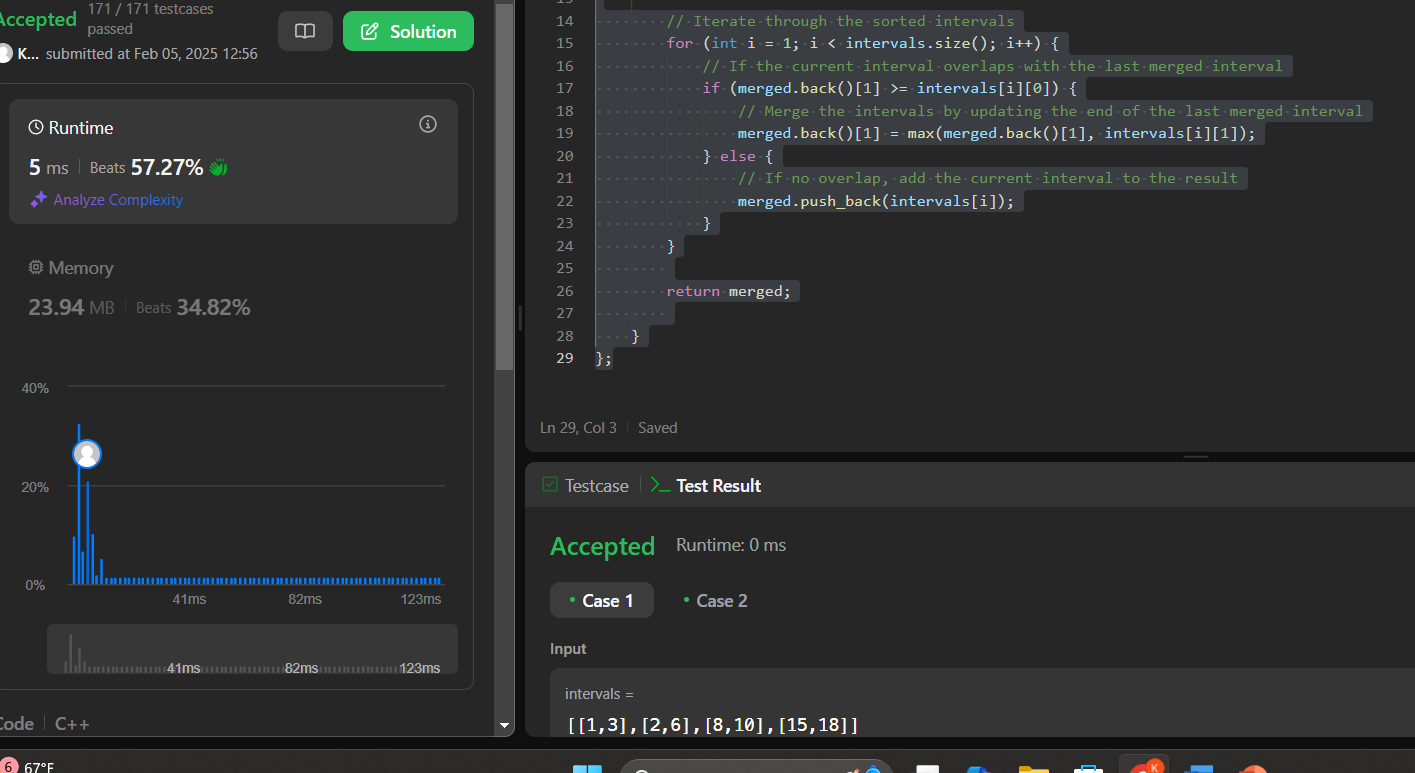
            }

        }

        return merged;

    }

};



**Question 11: FIND PEAK ELEMENT**

CODE:

class Solution {

public:

    int findPeakElement(vector<int>& nums) {

         int low = 0, high = nums.size() - 1;

        while (low < high) {

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

            if (nums[mid] > nums[mid + 1]) {

                // Peak is on the left side (including mid)

                high = mid;

            } else {

                // Peak is on the right side

                low = mid + 1;

            }

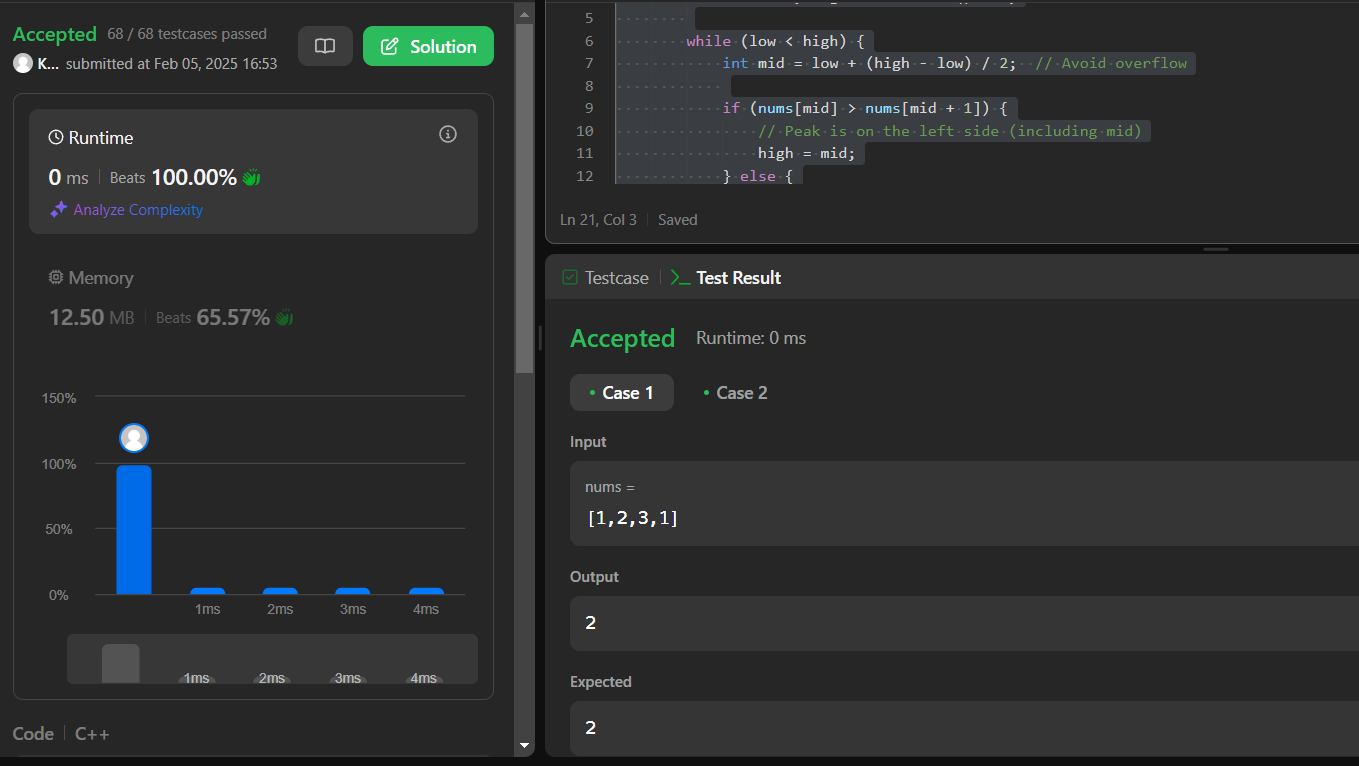
        }

        return low;  // 'low' will be the peak index

    }

};

OUTPUT:



**Question 12: FIRST BAD VERSION:**

CODE:

class Solution {

public:

    int firstBadVersion(int n) {

        int low = 1, high = n;

        while (low < high) {

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

            if (isBadVersion(mid)) {

                high = mid;  // The first bad version is at mid or before

            } else {

                low = mid + 1;  // Move right

            }

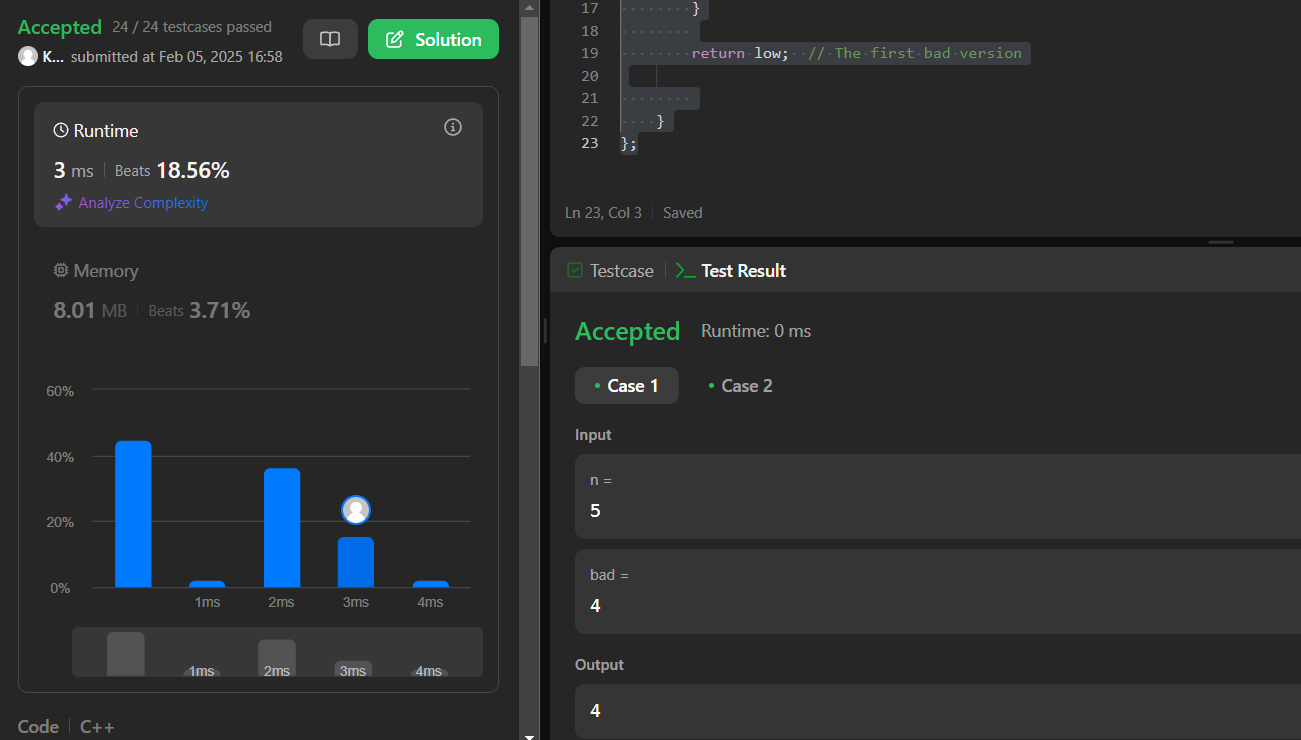
        }

        return low;  // The first bad version

    }

};

OUTPUT:



**QUESTION 13: REVERSE PAIRS**

**CODE:**

class Solution {

public:

    int reversePairs(vector<int>& nums) {

        return mergeSort(nums, 0, nums.size() - 1);

    }

private:

    int mergeSort(vector<int>& nums, int left, int right) {

        if (left >= right) return 0;  // Base case: single element

        int mid = left + (right - left) / 2;

        int count = mergeSort(nums, left, mid) + mergeSort(nums, mid + 1, right);

        // Count cross pairs (nums[i] > 2 \* nums[j])

        count += countReversePairs(nums, left, mid, right);

        // Merge the two halves

        merge(nums, left, mid, right);

        return count;

    }

    int countReversePairs(vector<int>& nums, int left, int mid, int right) {

        int count = 0, j = mid + 1;

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

            while (j <= right && nums[i] > 2LL \* nums[j]) {

                j++;  // Find the first valid j

            }

            count += (j - (mid + 1));

        }

        return count;

    }

    void merge(vector<int>& nums, int left, int mid, int right) {

        vector<int> temp;

        int i = left, j = mid + 1;

        while (i <= mid && j <= right) {

            if (nums[i] <= nums[j]) temp.push\_back(nums[i++]);

            else temp.push\_back(nums[j++]);

        }

        while (i <= mid) temp.push\_back(nums[i++]);

        while (j <= right) temp.push\_back(nums[j++]);

        // Copy sorted elements back

        for (int k = 0; k < temp.size(); k++) {

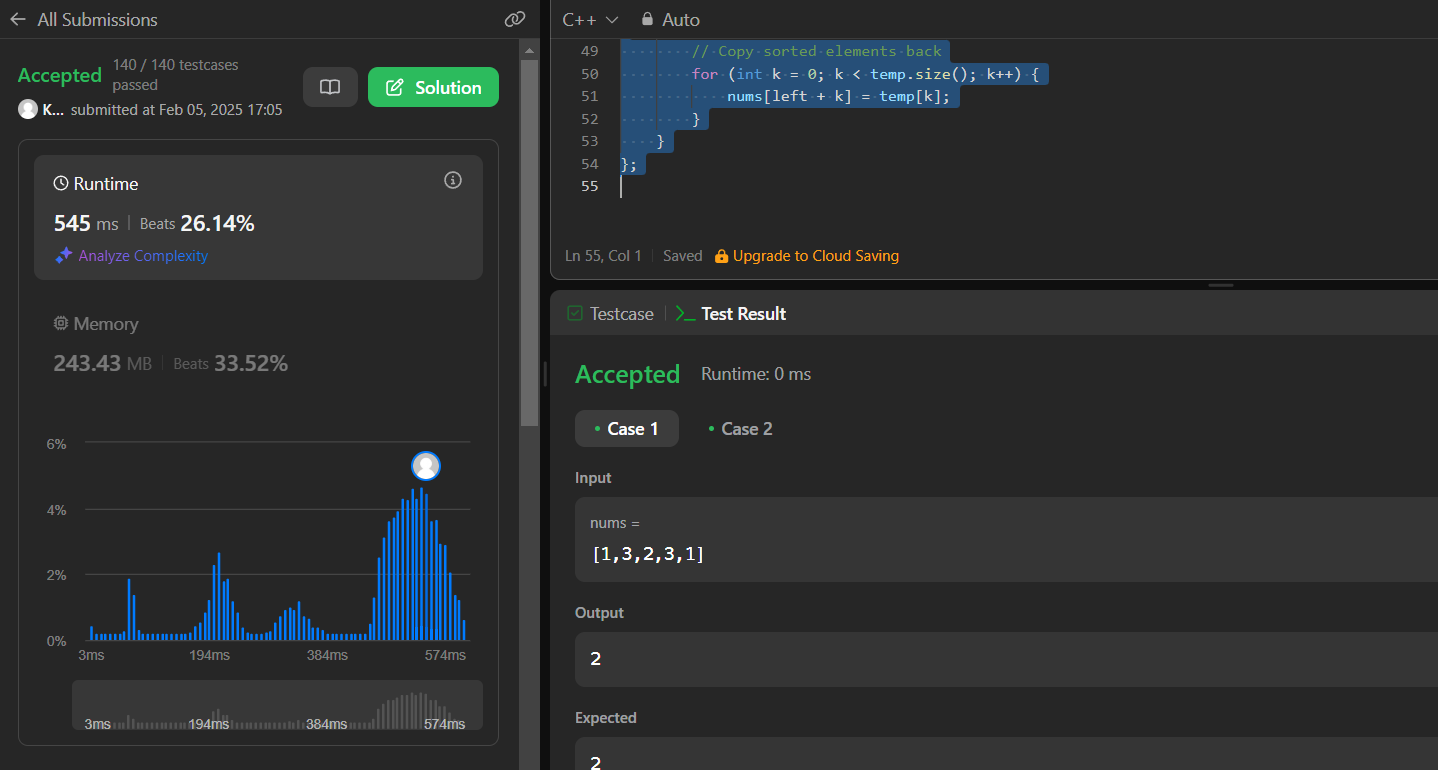
            nums[left + k] = temp[k];

        }

    }

};

**OUTPUT:**

****

**Question 14: BEAUTIFUL ARRAY:**

**CODE:**

class Solution {

public:

    vector<int> beautifulArray(int n) {

         vector<int> result = {1}; // Start with base case

        while (result.size() < n) {

            vector<int> temp;

            // Generate odd elements

            for (int num : result) {

                if (2 \* num - 1 <= n) temp.push\_back(2 \* num - 1);

            }

            // Generate even elements

            for (int num : result) {

                if (2 \* num <= n) temp.push\_back(2 \* num);

            }

            result = temp;

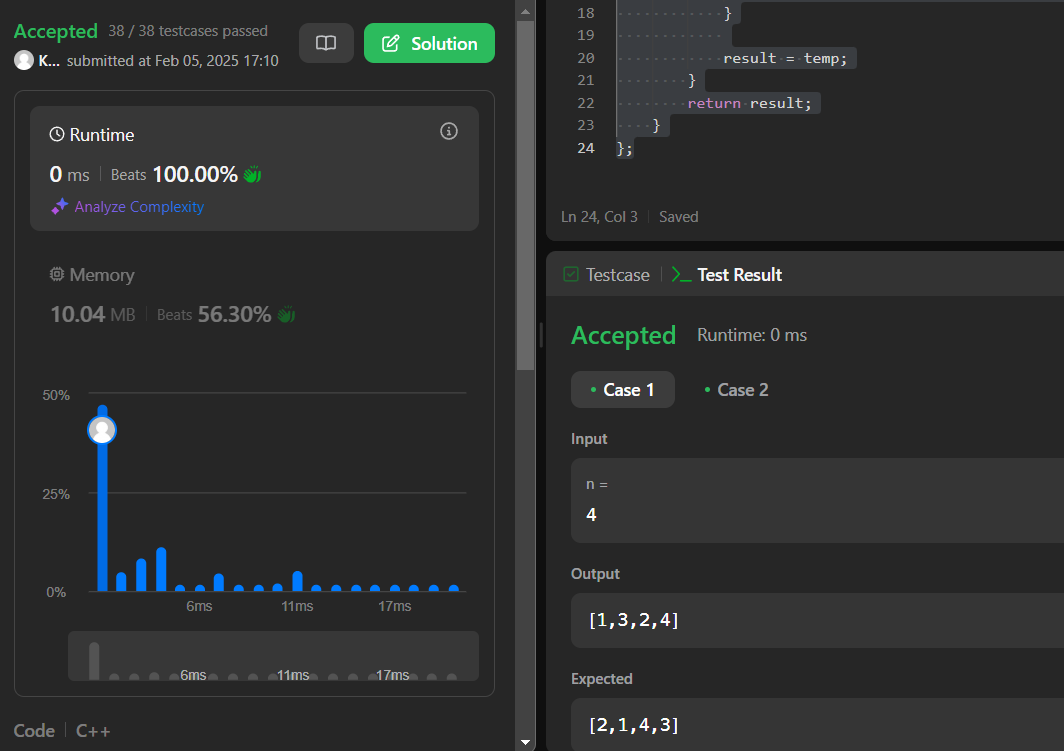
        }

        return result;

    }

};

**OUTPUT:**

****

**Question 15: THE SKYLINE PROBLEM**

**CODE:**

#include <vector>

#include <set>

#include <algorithm>

using namespace std;

class Solution {

public:

    vector<vector<int>> getSkyline(vector<vector<int>>& buildings) {

        vector<pair<int, int>> events; // Stores critical points

        // Step 1: Convert buildings into events

        for (auto& b : buildings) {

            events.emplace\_back(b[0], -b[2]); // Start event (-height for max heap)

            events.emplace\_back(b[1], b[2]);  // End event

        }

        // Step 2: Sort events

        sort(events.begin(), events.end(), [](const pair<int, int>& a, const pair<int, int>& b) {

            if (a.first == b.first) return a.second < b.second; // Process start (-h) before end (+h)

            return a.first < b.first;

        });

        // Step 3: Sweep line with max heap

        multiset<int> heights = {0}; // Start with ground level

        vector<vector<int>> result;

        int prevHeight = 0;

        for (auto& event : events) {

            int x = event.first, h = event.second;

            if (h < 0) heights.insert(-h); // Start event: insert height

            else heights.erase(heights.find(h)); // End event: remove height

            int currHeight = \*heights.rbegin(); // Get max height

            if (currHeight != prevHeight) { // If height changes, add key point

                result.push\_back({x, currHeight});

                prevHeight = currHeight;

            }

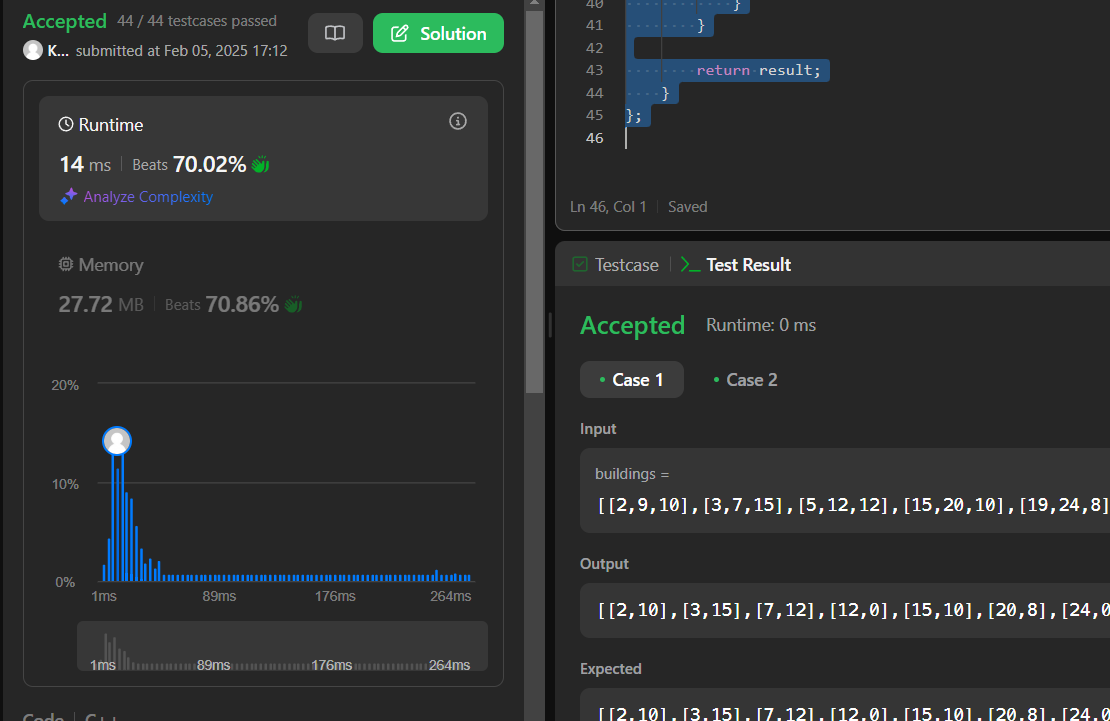
        }

        return result;

    }

};

**OUTPUT:**

****