Name: Kashish Chhimwal

UID: 22BCS14983

Section: 605(B)

Question1: Convert sorted array to binary search tree.

Code:

class Solution {

public:

    TreeNode\* sortedArrayToBST(vector<int>& nums) {

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

    }

    TreeNode\* constructBST(vector<int>& nums, int left, int right) {

        if (left > right) return nullptr;

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

        TreeNode\* root = new TreeNode(nums[mid]);

        root->left = constructBST(nums, left, mid - 1);

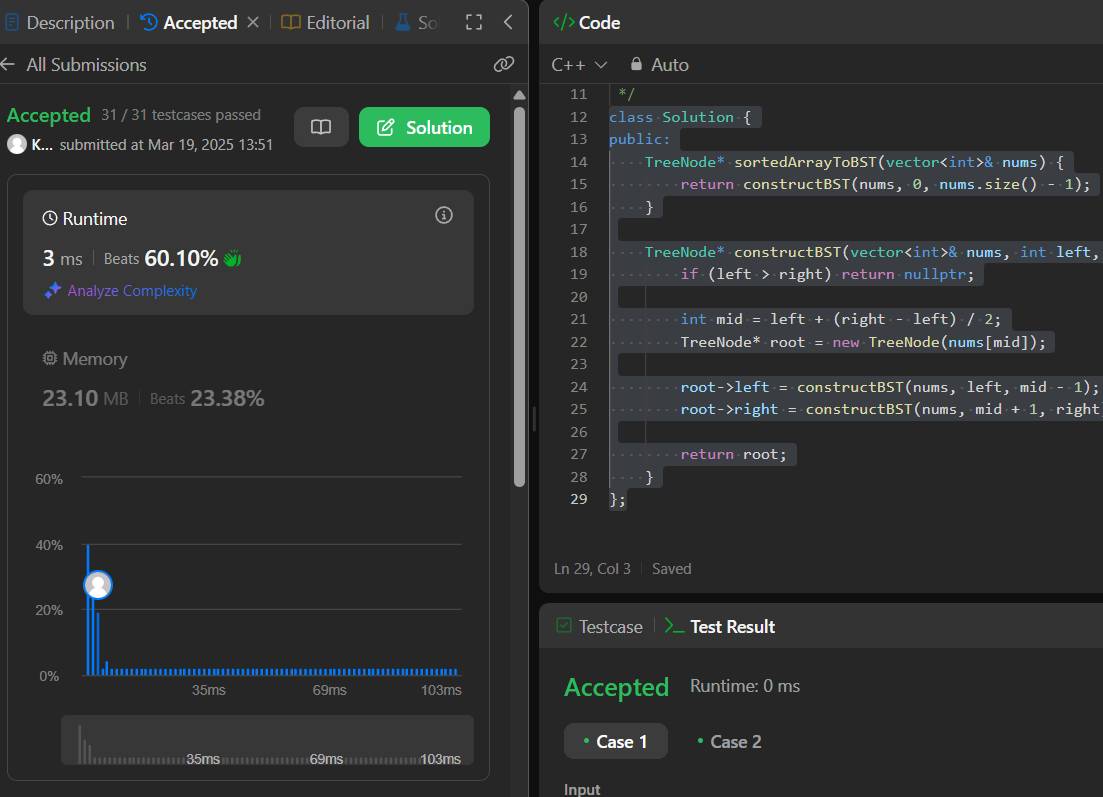
        root->right = constructBST(nums, mid + 1, right);

        return root;

    }

};

Output:



Question 2: Number of 1 Bits

Code:

class Solution {

public:

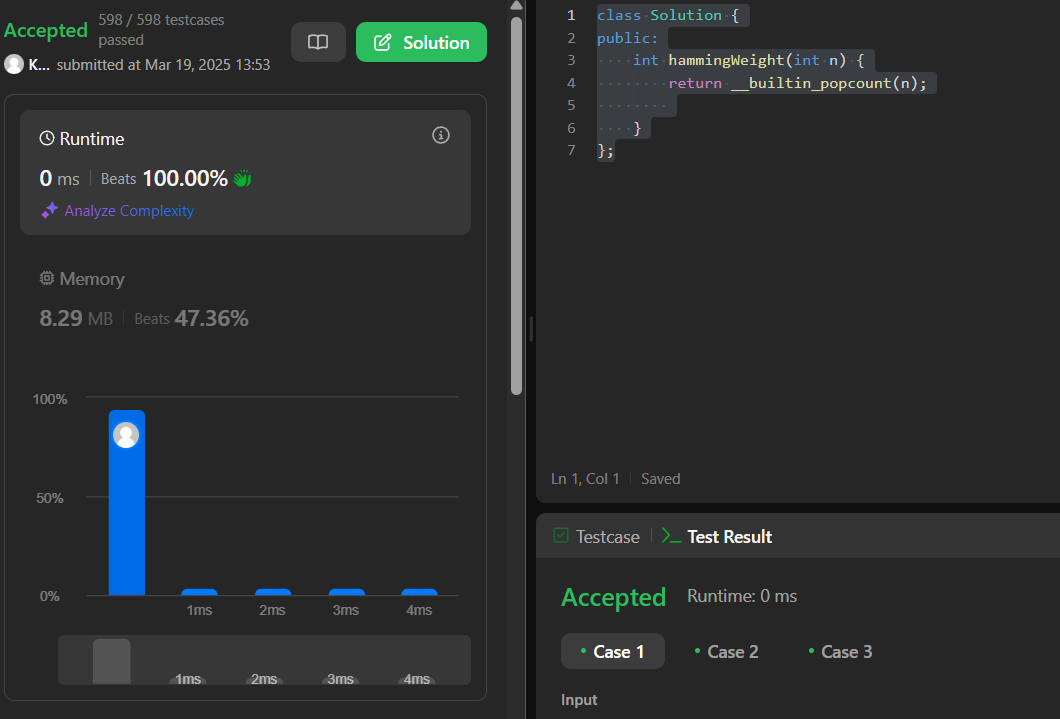
    int hammingWeight(int n) {

        return \_\_builtin\_popcount(n);

    }

};

Output:



Question 3: Sort An array

Code:

class Solution {

public:

    void quickSort(vector<int>& nums, int low, int high) {

        if (low < high) {

            if (high - low < 10) { // Use insertion sort for small arrays

                insertionSort(nums, low, high);

                return;

            }

            int pivotIndex = randomizedPartition(nums, low, high);

            quickSort(nums, low, pivotIndex - 1);

            quickSort(nums, pivotIndex + 1, high);

        }

    }

    int randomizedPartition(vector<int>& nums, int low, int high) {

        int randomIndex = low + rand() % (high - low + 1);

        swap(nums[randomIndex], nums[high]); // Move random pivot to the end

        return partition(nums, low, high);

    }

    int partition(vector<int>& nums, int low, int high) {

        int pivot = nums[high];

        int i = low - 1;

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

            if (nums[j] < pivot) {

                i++;

                swap(nums[i], nums[j]);

            }

        }

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

        return i + 1;

    }

    void insertionSort(vector<int>& nums, int low, int high) {

        for (int i = low + 1; i <= high; i++) {

            int key = nums[i];

            int j = i - 1;

            while (j >= low && nums[j] > key) {

                nums[j + 1] = nums[j];

                j--;

            }

            nums[j + 1] = key;

        }

    }

    vector<int> sortArray(vector<int>& nums) {

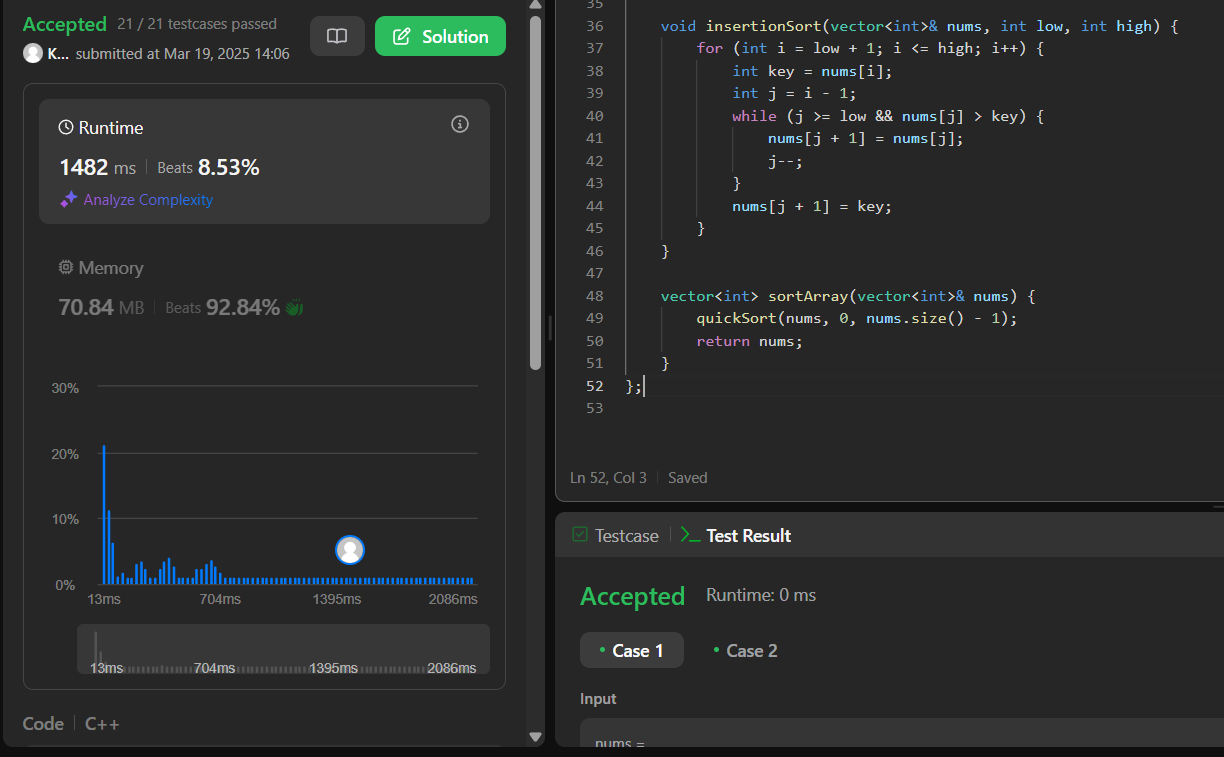
        quickSort(nums, 0, nums.size() - 1);

        return nums;

    }

};

Output:



Question 4: Maximum Subarray

Code:

class Solution {

public:

    int maxSubArray(vector<int>& nums) {

        int maxSum = INT\_MIN, currentSum = 0;

    for (int num : nums) {

        currentSum = max(num, currentSum + num);

        maxSum = max(maxSum, currentSum);

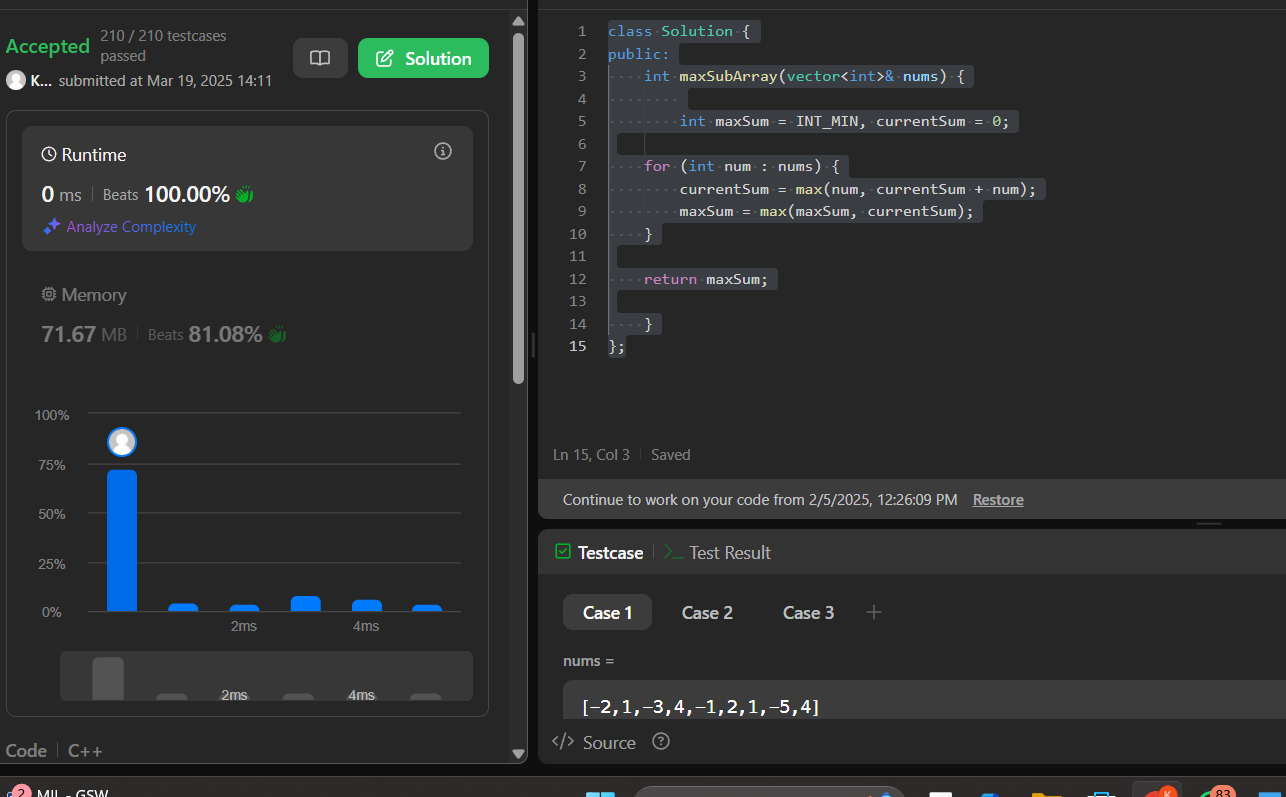
    }

    return maxSum;

    }

};

Output:



Question 5: 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;

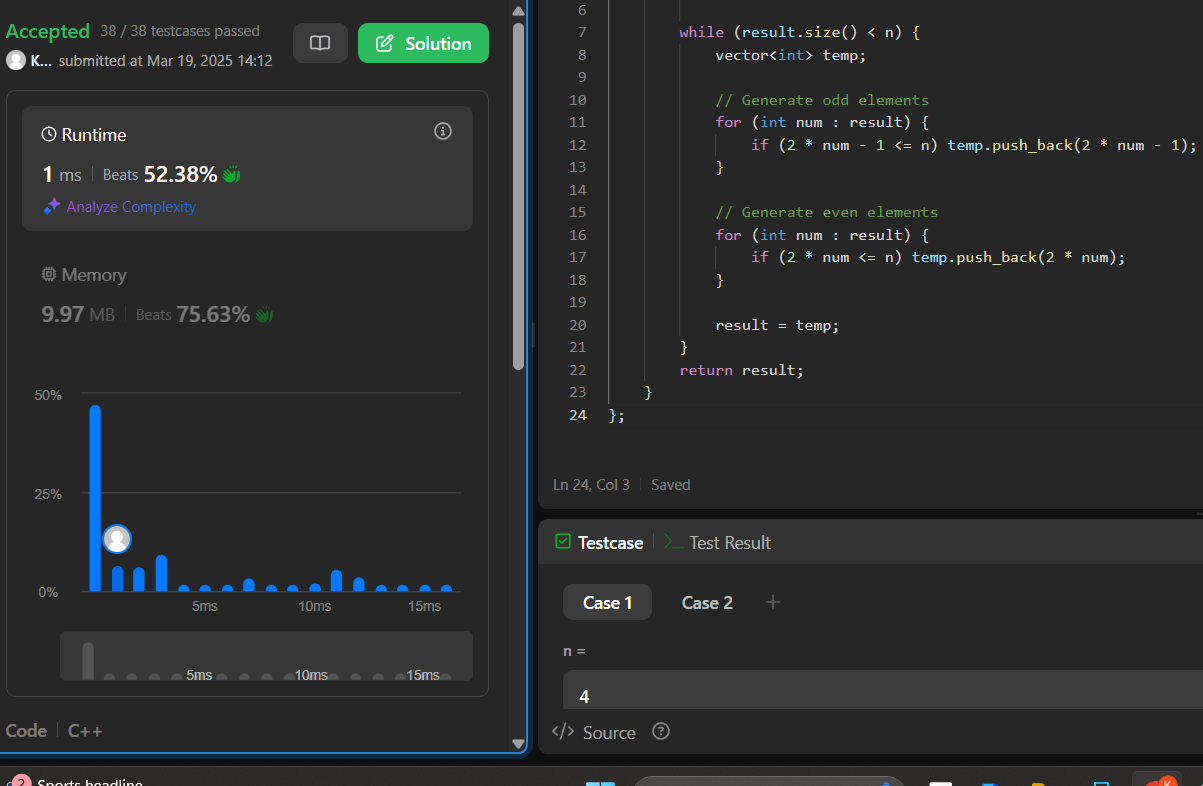
        }

        return result;

    }

};

Output:



Question 7: 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:

