ADVANCED PROGRAMMING LAB ASSIGNMENT

1. LONGEST NICE SUBSTRING (1763)

class Solution {

public:

    string longestNiceSubstring(string s) {

        if (s.length() < 2) return "";

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

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

            char c = s[i];

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

                continue;

            }

            // Split the string at the bad character and solve recursively

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

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

            return (left.length() >= right.length()) ? left : right;

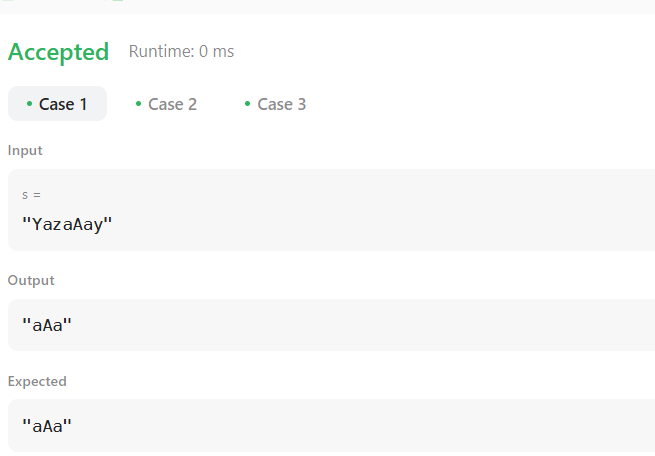
        }

        return s; // If all characters are nice, return the whole string

    }

};

Output-



1. REVERSE BITS (190)

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);  // Shift result left and add the last bit of n

            n >>= 1;  // Shift n right to process the next bit

        }

        return result;

    }

};

Output-



1. NUMBER OF 1 BITS (191)

class Solution {

public:

    int hammingWeight(int n) {

        int count = 0;

        while (n) {

            count += (n & 1);  // Add the last bit of n

            n >>= 1;            // Shift n right by 1 to check the next bit

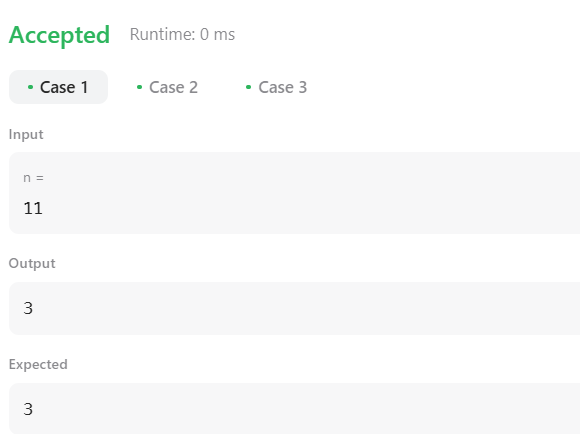
        }

        return count;

    }

};

Output-



1. MAXIMUM SUBARRAY (53)

class Solution {

public:

    int maxSubArray(vector<int>& nums) {

        int currentSum = nums[0];  // Initialize currentSum with the first element

        int maxSum = nums[0];      // Initialize maxSum with the first element

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

            // Update currentSum: either start new subarray or extend the current one

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

            // Update maxSum if currentSum is larger

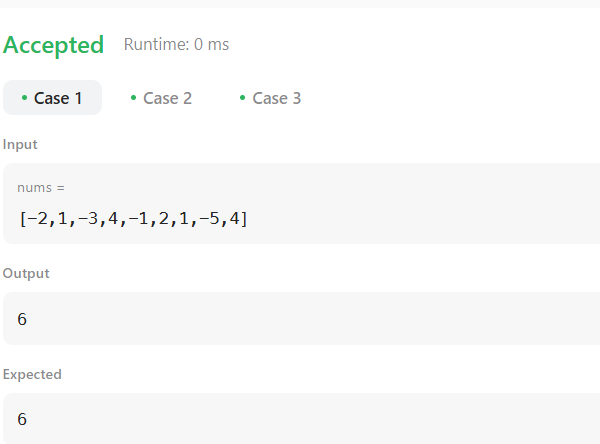
            maxSum = max(maxSum, currentSum);

        }

        return maxSum;

    }

};

Output-

1. SEARCH A 2D MATRIX II (240)

class Solution {

public:

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

        int row = 0;

        int col = matrix[0].size() - 1; // Start from the top-right corner

        while (row < matrix.size() && col >= 0) {

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

                return true;  // Target found

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

                col--;  // Move left

            } else {

                row++;  // Move down

            }

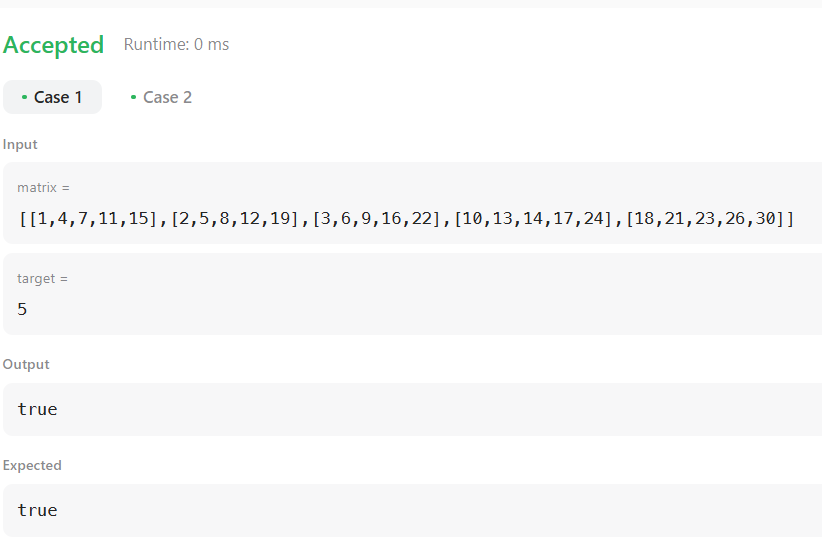
        }

        return false;  // Target not found

    }

};

Output-



1. SUPER POW (372)

class Solution {

public:

int powMod(int a, int b, int mod) {

        int result = 1;

        a = a % mod;  // Handle the case where a is greater than mod

        while (b > 0) {

            if (b % 2 == 1) {  // If b is odd, multiply a with result

                result = (result \* a) % mod;

            }

            a = (a \* a) % mod;  // Square the base

            b /= 2;  // Reduce b by half

        }

        return result;

    }

    int superPow(int a, vector<int>& b) {

        int mod = 1337;

        a = a % mod;  // Take a modulo 1337

        int result = 1;

        // Process each digit in array b

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

            // Update the result with current digit in b

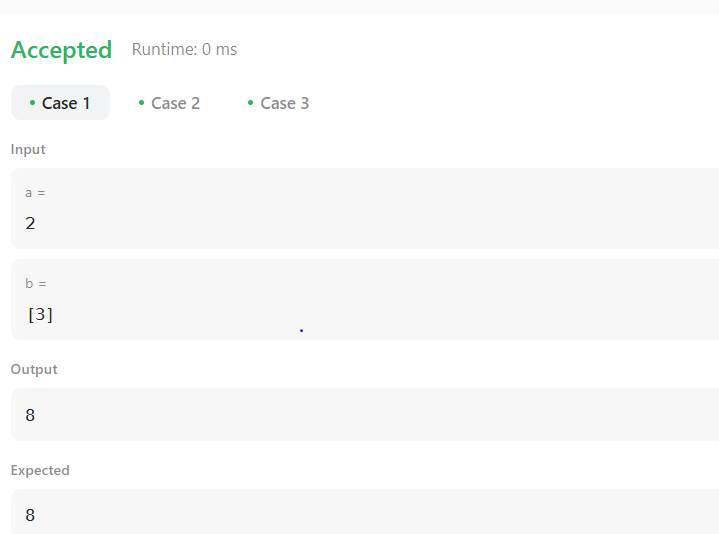
            result = (powMod(result, 10, mod) \* powMod(a, b[i], mod)) % mod;

        }

        return result;

    }};

Output-



1. BEAUTIFUL ARRAY (932)

class Solution {

public:

    vector<int> beautifulArray(int n) {

        vector<int> arr = {1};

        // Loop until we build the array for the required size n

        while (arr.size() < n) {

            vector<int> temp;

            // Add odd numbers from the current array

            for (int num : arr) {

                if (num \* 2 - 1 <= n) {

                    temp.push\_back(num \* 2 - 1);

                }

            }

            // Add even numbers from the current array

            for (int num : arr) {

                if (num \* 2 <= n) {

                    temp.push\_back(num \* 2);

                }

            }

            // Update arr to be the new temp array

            arr = temp;

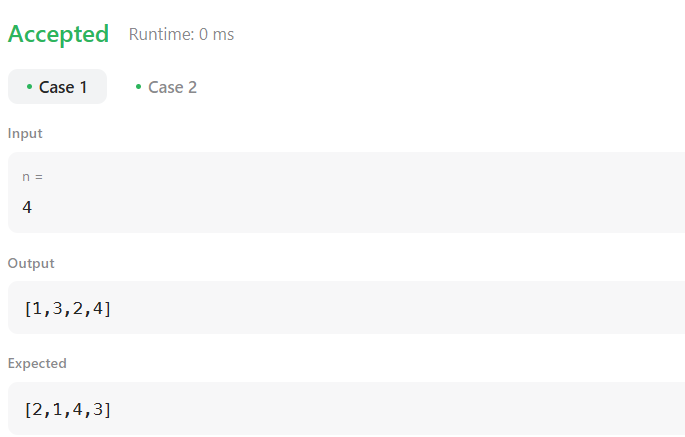
        }

        return arr;

    }

};

Output-



1. MERGE SORTED ARRAY (88)

class Solution {

public:

    void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {

         int i = m + n - 1;

        int i1 = m - 1;  // Last valid index of nums1

        int i2 = n - 1;  // Last index of nums2

        // Compare elements from the back and fill nums1

        while (i1 >= 0 && i2 >= 0) {

            if (nums1[i1] > nums2[i2]) {

                nums1[i] = nums1[i1];

                i1--;

            } else {

                nums1[i] = nums2[i2];

                i2--;

            }

            i--;

        }

        // If any elements left in nums2, copy them to nums1

        while (i2 >= 0) {

            nums1[i] = nums2[i2];

            i--;

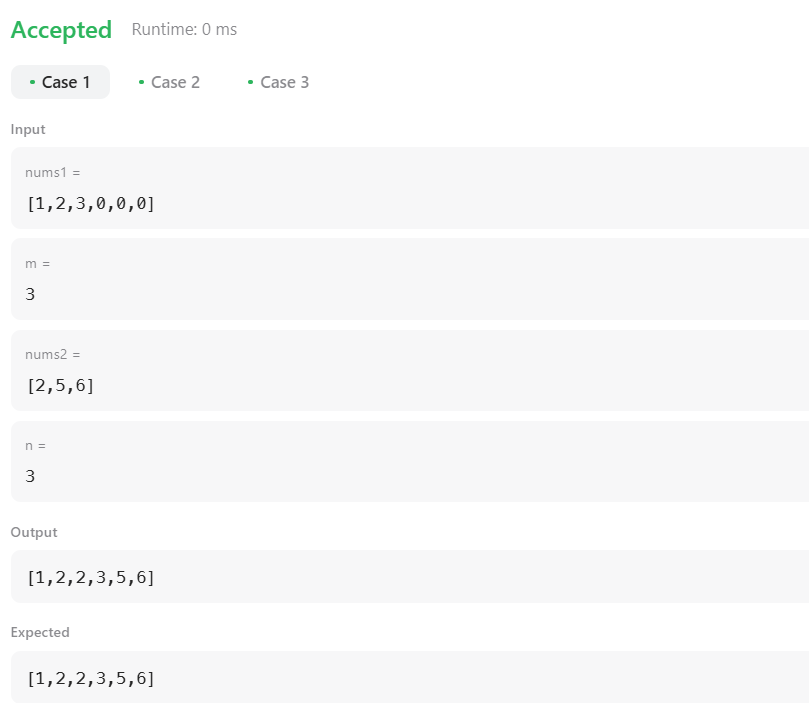
            i2--;

        }

    }

};

Output-



1. FIRST BAD VERSION (278)

class Solution {

public:

    int firstBadVersion(int n) {

         int left = 1, right = n;

        while (left < right) {

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

            if (isBadVersion(mid)) {

                right = mid;  // Narrow down the search to the left half

            } else {                left = mid + 1;  // Narrow down the search to the right half

            }

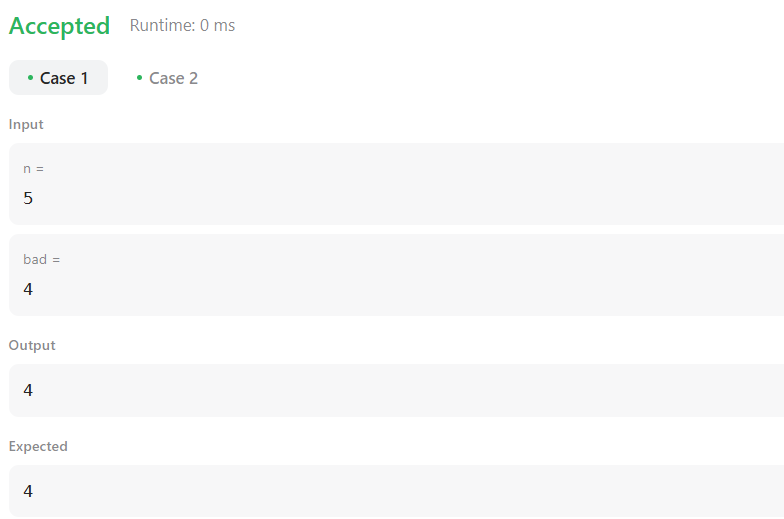
        }

        return left;  // Left and right will converge to the first bad version

    }

};

Output-



1. SORT COLORS (75)

class Solution {

public:

    void sortColors(vector<int>& nums) {

        int count0 = 0, count1 = 0, count2 = 0;

        // Count occurrences

        for (int num : nums) {

            if (num == 0) count0++;

            else if (num == 1) count1++;

            else count2++;

        }

        // Overwrite nums with sorted values

        int i = 0;

        while (count0--) nums[i++] = 0;

        while (count1--) nums[i++] = 1;

        while (count2--) nums[i++] = 2;

    }

};

Output-



1. TOP K FREQUENT ELEMENTS (347)

class Solution {

public:

    vector<int> topKFrequent(vector<int>& nums, int k) {

         unordered\_map<int, int> freqMap;

        // Step 1: Count frequency of each number

        for (int num : nums) {

            freqMap[num]++;

        }

        // Step 2: Create frequency buckets

        int n = nums.size();

        vector<vector<int>> buckets(n + 1);  // Buckets from 0 to n

        for (auto& pair : freqMap) {

            buckets[pair.second].push\_back(pair.first);

        }

        // Step 3: Collect k most frequent elements from highest frequency buckets

        vector<int> result;

        for (int i = n; i >= 0 && result.size() < k; --i) {

            for (int num : buckets[i]) {

                result.push\_back(num);

                if (result.size() == k) return result;

            }

        }

        return result;

    }

};

Output-



1. Kth LARGEST ELEMENT IN AN ARRAY(215)

class Solution {

public:

    int findKthLargest(vector<int>& nums, int k) {

        priority\_queue<int, vector<int>, greater<int>> minHeap;

        for (int num : nums) {

            minHeap.push(num);

            if (minHeap.size() > k) {

                minHeap.pop();  // Remove smallest element

            }

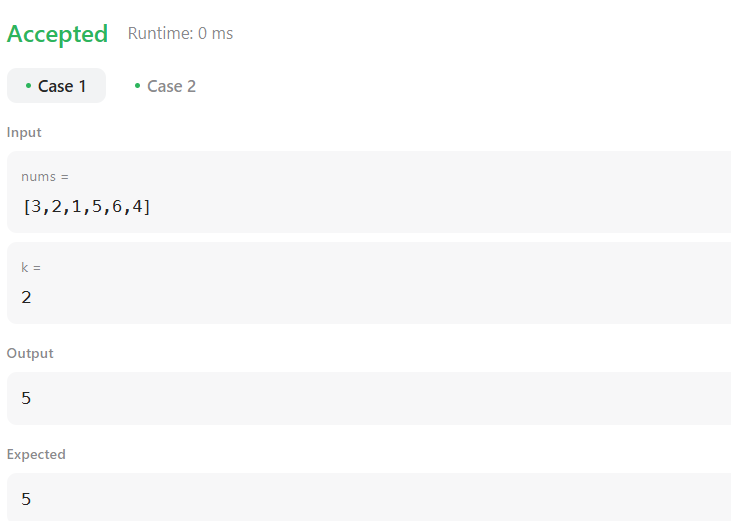
        }

        return minHeap.top();

    }

};

Output-



1. FIND PEAK ELEMENT (162)

class Solution {

public:

    int findPeakElement(vector<int>& nums) {

        int n = nums.size();

        if (n == 1) return 0;  // Only one element, it's the peak

        for (int i = 0; i < n; i++) {

            if ((i == 0 || nums[i] > nums[i - 1]) && (i == n - 1 || nums[i] > nums[i + 1])) {

                return i;

            }

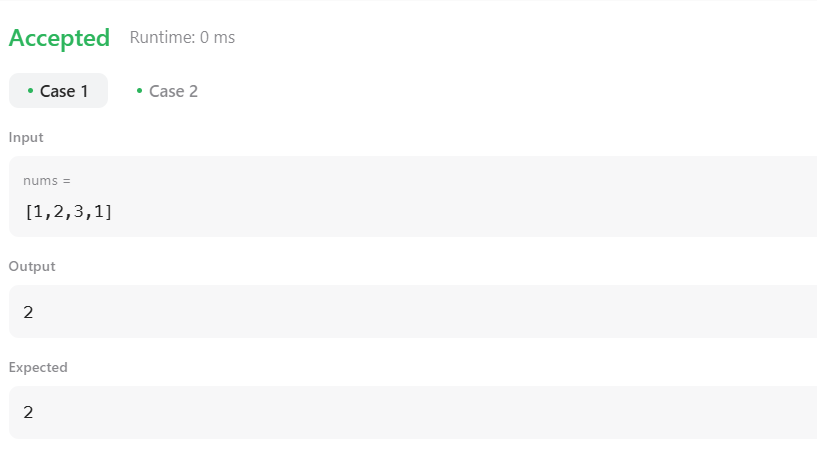
        }

        return -1;

    }

};

Output-



1. MERGE INTERVALS (56)

class Solution {

public:

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

        if (intervals.empty()) return {};  // Edge case: No intervals

        // Step 1: Sort intervals based on the start time

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

        vector<vector<int>> merged;

        for (auto& interval : intervals) {

            // If merged is empty OR no overlap with the last interval, add new interval

            if (merged.empty() || merged.back()[1] < interval[0]) {

                merged.push\_back(interval);

            } else {

                // Merge overlapping intervals by updating the end time

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

            }

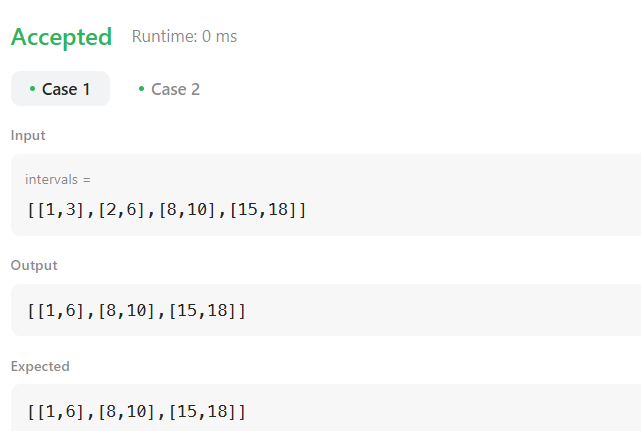
        }

        return merged;

    }

};

Output-



1. SEARCH IN ROTATED ARRAY (33)

class Solution {

public:

    int search(vector<int>& nums, int target) {

        int left = 0, right = nums.size() - 1;

        while (left <= right) {

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

            // Check if mid is the target

            if (nums[mid] == target) return mid;

            // Determine which part is sorted

            if (nums[left] <= nums[mid]) { // Left half is sorted

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

                    right = mid - 1; // Search in left sorted half

                } else {

                    left = mid + 1; // Search in right half

                }

            } else { // Right half is sorted

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

                    left = mid + 1; // Search in right sorted half

                } else {

                    right = mid - 1; // Search in left half

                }

            }

        }

        return -1; // Target not found

    }

};

Output-

