**Experiment 4**

**Name: Sanju Arora Branch: BE in IT**

**Uid: 22BET10059 Semester: 6TH**

**Section/Group: 22BET\_IO1\_702/A. Date Of Performance: 14/2/2025**

**Subject Name: Advanced Programming Lab-2 Subject Code: 22ITT-367**

* **Aim:**

1. To find the longest substring where every character appears in both lowercase and uppercase.
2. To reverse the bits of a given 32-bit unsigned integer.
3. To count the number of 1 bits in a given 32-bit integer (Hamming Weight).
4. To find the contiguous subarray (within a one-dimensional array) that has the largest sum (Kadane’s Algorithm / Divide & Conquer).
5. To search for a target integer in a matrix where each row is sorted left to right and each column is sorted top to bottom.
6. To compute (a^b) % 1337 efficiently using modular exponentiation.
7. To construct an array that avoids arithmetic sequences while keeping distinct integers.
8. To determine the skyline formed by a given set of buildings using a Divide & Conquer or Heap-based approach.
9. To count the number of reverse pairs (i, j) where i < j and nums[i] > 2 \* nums[j], using a **Merge Sort-based approach.**
10. To find the longest increasing subsequence with a maximum difference constraint using a **Segment Tree.**

* **Code:**

1. **Longest Nice Substring**

class Solution {

public String longestNiceSubstring(String s) {

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

Set<Character> set = new HashSet<>();

for (char c : s.toCharArray()) {

set.add(c);

}

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

char c = s.charAt(i);

if(set.contains(Character.toUpperCase(c))&&set.contains(Character.toLowerCase(c))) {

continue; }

String left = longestNiceSubstring(s.substring(0, i));

String right = longestNiceSubstring(s.substring(i + 1));

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

}

return s;

}

}

1. **Reverse Bits**

class Solution {

// Function to reverse bits of a given 32 bits unsigned integer

public int reverseBits(int n) {

int result = 0;

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

result <<= 1; // Shift result to the left by 1

result |= (n & 1); // Extract the last bit of n and add it to result

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

}

return result;

}

}

1. **Number of 1 Bits**

class Solution {

// Function to count the number of 1 bits in a given 32 bits unsigned integer

public int hammingWeight(int n) {

int count = 0;

while (n != 0) {

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

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

}

return count;

}

}

1. **Maximum Subarray**

class Solution {

// Function to find the maximum subarray sum using Kadane's Algorithm

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;

}

}

1. **Search a 2D Matrix**

class Solution {

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

int rows = matrix.length;

int cols = matrix[0].length;

int row = 0, col = cols - 1;

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

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

return true;

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

col--;

} else {

row++;

}

}

return false;

}

}

1. **Super Pow**

class Solution {

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

int mod = 1337;

a %= mod;

return powerMod(a, b, mod);

}

private int powerMod(int a, int[] b, int mod) {

int result = 1;

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

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

}

return result; }

private int power(int x, int y, int mod) {

int result = 1;

while (y > 0) {

if (y % 2 == 1) {

result = result \* x % mod;

}

x = x \* x % mod;

y /= 2;

}

return result;

}

}

1. **Beautiful Array**

class Solution {

public int[] beautifulArray(int n) {

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

result.add(1);

while (result.size() < n) {

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

for (int num : result) {

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

temp.add(num \* 2 - 1); // Odd numbers

}

}

for (int num : result) {

if (num \* 2 <= n) {

temp.add(num \* 2); // Even numbers

}

}

result = temp;

}

return result.stream().mapToInt(i -> i).toArray();

}

}

1. **The Skyline Problem**

import java.util.\*;

class Solution {

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

if (buildings == null || buildings.length == 0) return new ArrayList<>();

return divideAndConquer(buildings, 0, buildings.length - 1);

}

private List<List<Integer>> divideAndConquer(int[][] buildings, int left, int right) {

if (left == right) {

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

result.add(Arrays.asList(buildings[left][0], buildings[left][2]));

result.add(Arrays.asList(buildings[left][1], 0));

return result;

}

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

List<List<Integer>> leftSkyline = divideAndConquer(buildings, left, mid);

List<List<Integer>> rightSkyline = divideAndConquer(buildings, mid + 1, right);

return mergeSkylines(leftSkyline, rightSkyline);

}

private List<List<Integer>> mergeSkylines(List<List<Integer>> left, List<List<Integer>> right) {

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

int h1 = 0, h2 = 0, x = 0, height = 0;

int i = 0, j = 0;

while (i < left.size() && j < right.size()) {

List<Integer> point1 = left.get(i);

List<Integer> point2 = right.get(j);

if (point1.get(0) < point2.get(0)) {

x = point1.get(0);

h1 = point1.get(1);

i++;

} else if (point1.get(0) > point2.get(0)) {

x = point2.get(0);

h2 = point2.get(1);

j++;

} else {

x = point1.get(0);

h1 = point1.get(1);

h2 = point2.get(1);

i++;

j++;

}

int maxHeight = Math.max(h1, h2);

if (merged.isEmpty() || merged.get(merged.size() - 1).get(1) != maxHeight) {

merged.add(Arrays.asList(x, maxHeight));

}

}

while (i < left.size()) merged.add(left.get(i++));

while (j < right.size()) merged.add(right.get(j++));

return merged;

}

}

1. **Reverse Pairs**

class Solution {

public int reversePairs(int[] nums) {

if (nums == null || nums.length == 0) return 0;

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

}

private int mergeSort(int[] nums, int left, int right) {

if (left >= right) return 0;

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

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

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

merge(nums, left, mid, right);

return count;

}

private int countPairs(int[] nums, int left, int mid, int right) {

int count = 0, j = mid + 1;

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

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

j++;

}

count += (j - (mid + 1));

}

return count;

}

private void merge(int[] nums, int left, int mid, int right) {

int[] temp = new int[right - left + 1];

int i = left, j = mid + 1, k = 0;

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

if (nums[i] <= nums[j]) {

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

} else {

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

}

}

while (i <= mid) temp[k++] = nums[i++];

while (j <= right) temp[k++] = nums[j++];

System.arraycopy(temp, 0, nums, left, temp.length);

}

}

1. **Longest Increasing Subsequence**

class SegmentTree {

int[] tree;

int n;

public SegmentTree(int size) {

n = size;

tree = new int[4 \* n];

}

public void update(int index, int value, int left, int right, int node) {

if (left == right) {

tree[node] = value;

return;

}

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

if (index <= mid) {

update(index, value, left, mid, 2 \* node + 1);

} else {

update(index, value, mid + 1, right, 2 \* node + 2);

}

tree[node] = Math.max(tree[2 \* node + 1], tree[2 \* node + 2]);

}

public int query(int start, int end, int left, int right, int node) {

if (start > right || end < left) return 0;

if (start <= left && right <= end) return tree[node];

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

return Math.max(query(start, end, left, mid, 2 \* node + 1), query(start, end, mid + 1, right, 2 \* node + 2));

}

}

class Solution {

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

int maxVal = 0;

for (int num : nums) maxVal = Math.max(maxVal, num);

SegmentTree segTree = new SegmentTree(maxVal + 1);

int maxLength = 0;

for (int num : nums) {

int bestPrev = segTree.query(Math.max(0, num - k), num - 1, 0, maxVal, 0);

int newLength = bestPrev + 1;

segTree.update(num, newLength, 0, maxVal, 0);

maxLength = Math.max(maxLength, newLength);

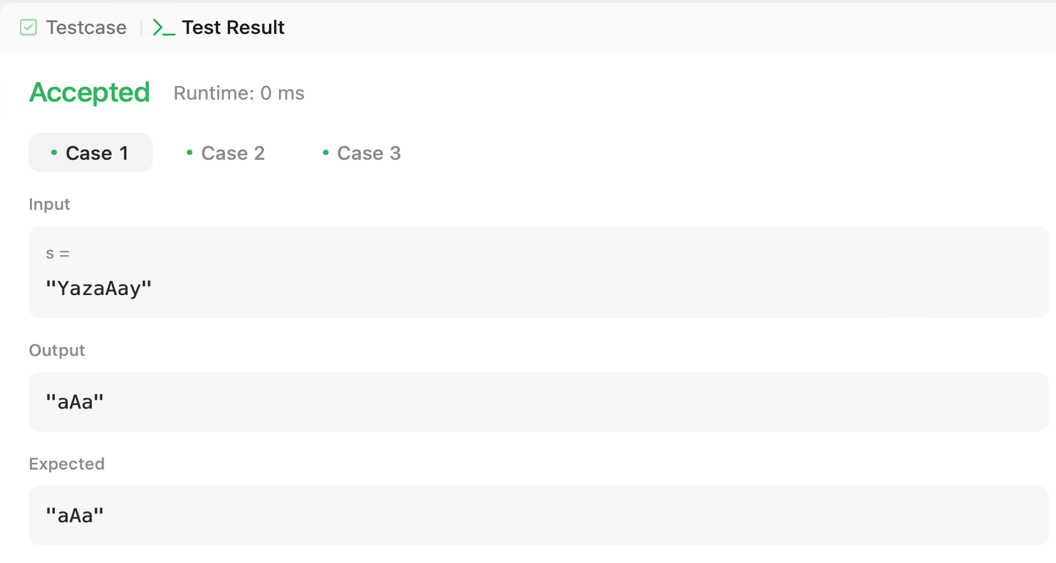
}

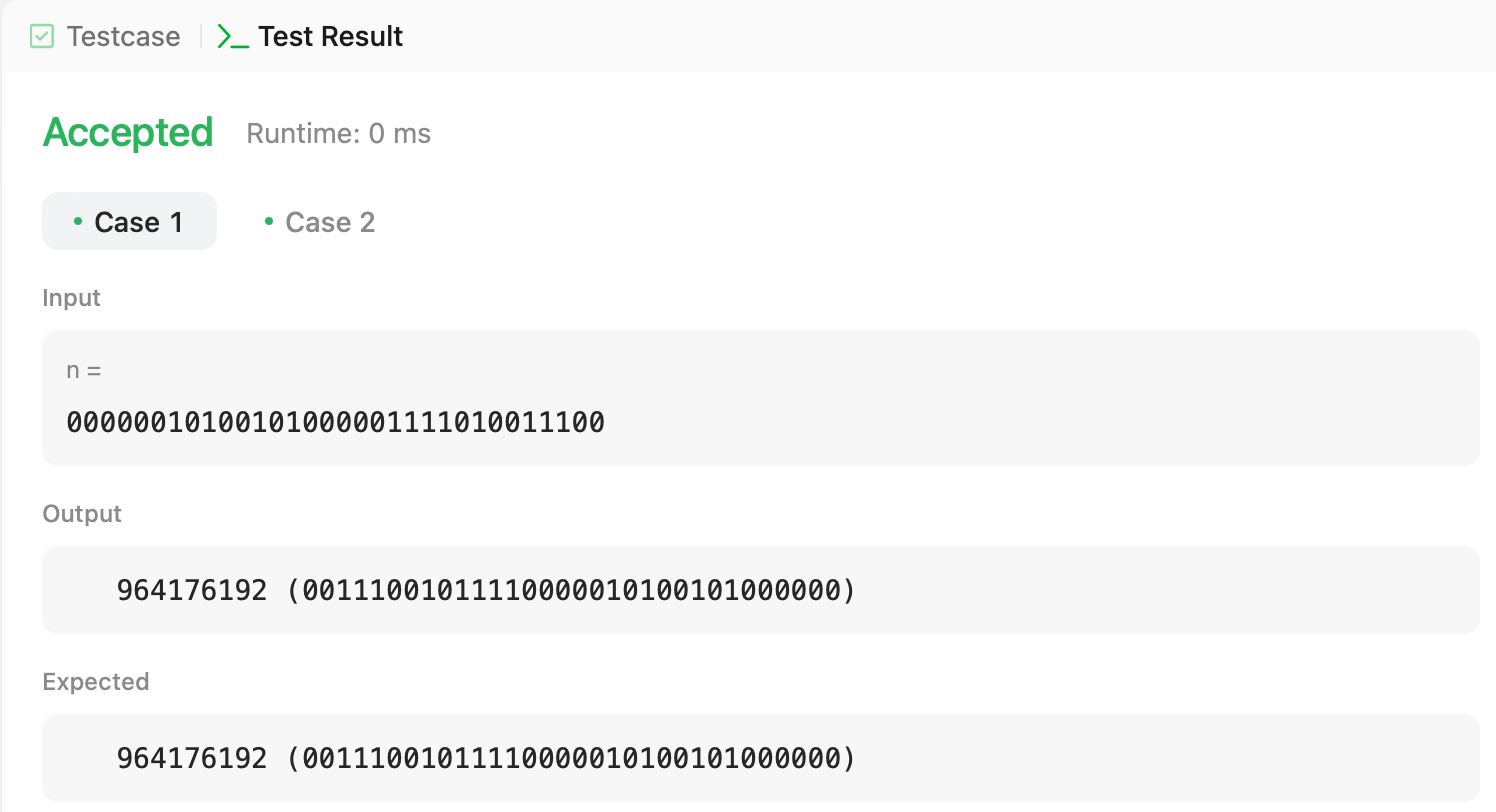
return maxLength;

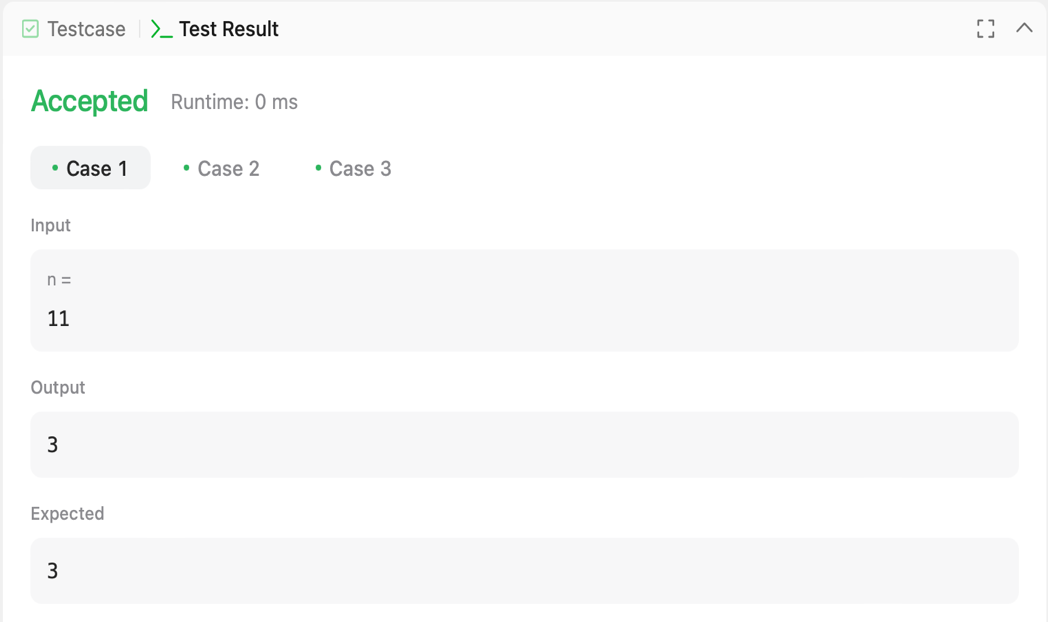
}

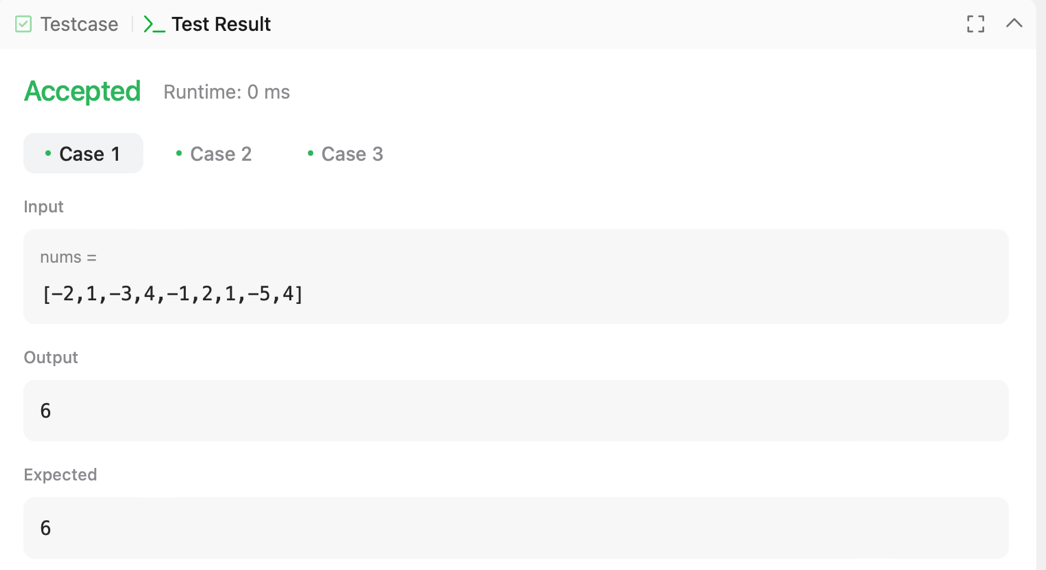
}

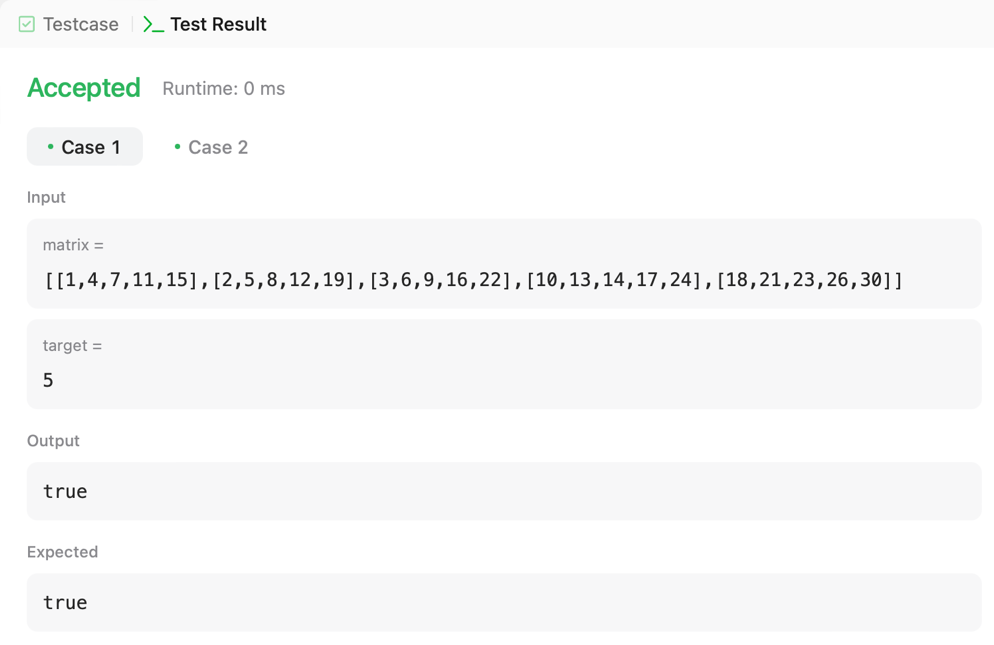
* **Output:**

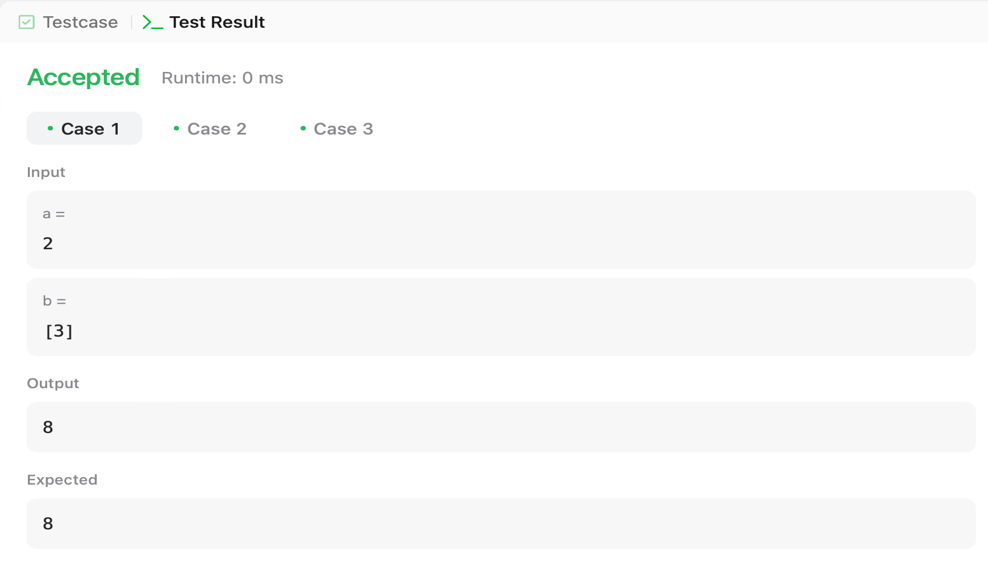
****

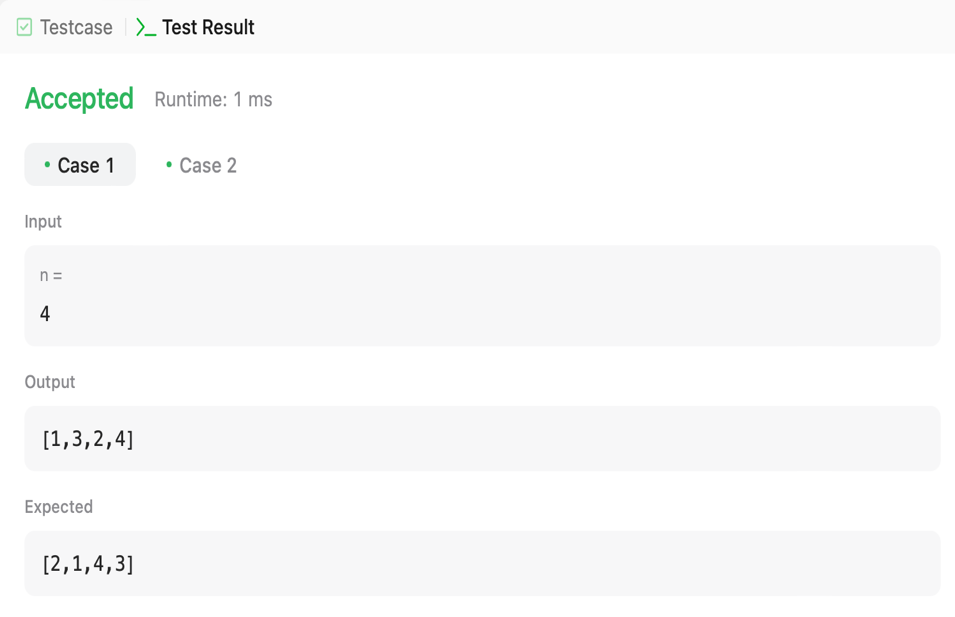
****

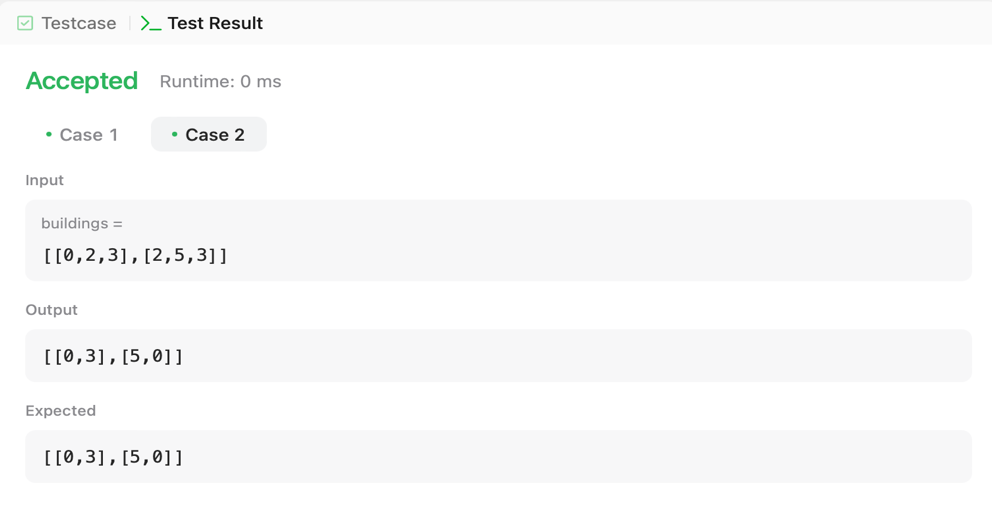
****

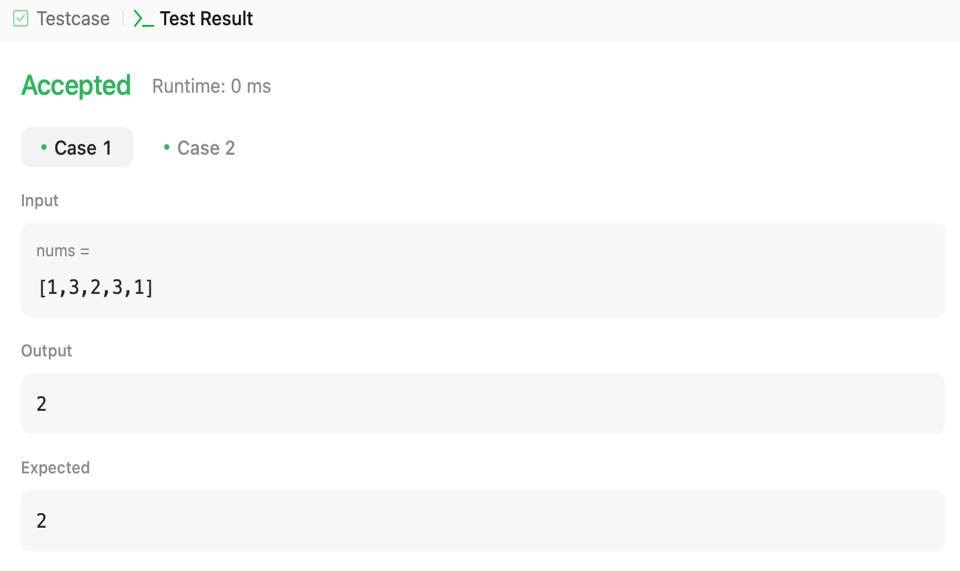
****

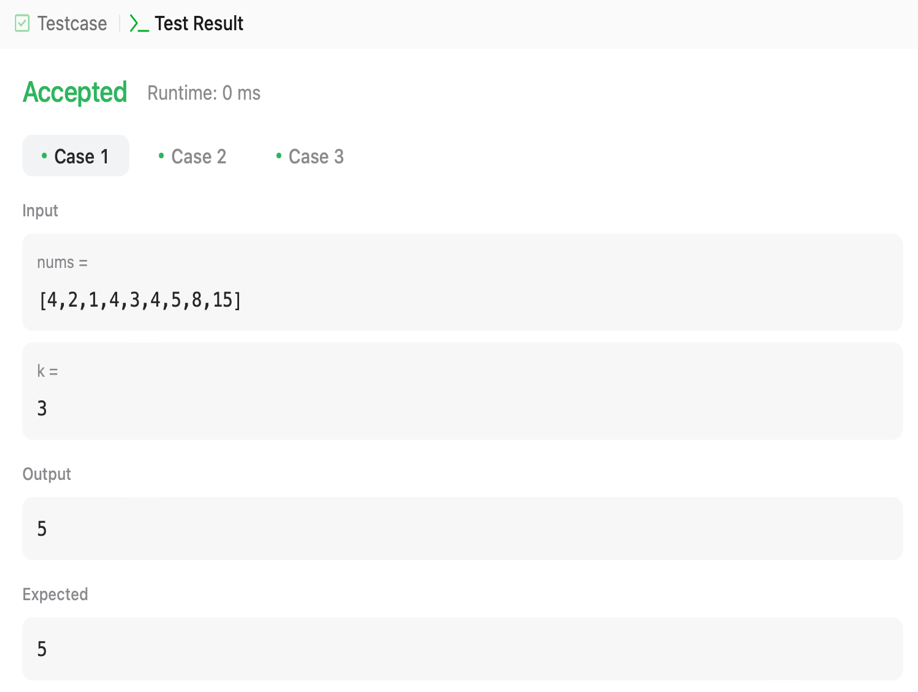
****

****

****

****

****

****