Experiment 7

Name: Arushi Butola **UID: 22BET10027**
Branch: IT **Section/Group: 22BET-702/B**
Semester: 6 **Date of Performance: 20-03-25**
Subject Name: Advanced Programming Lab-2 **Subject Code: 22ITP-351**

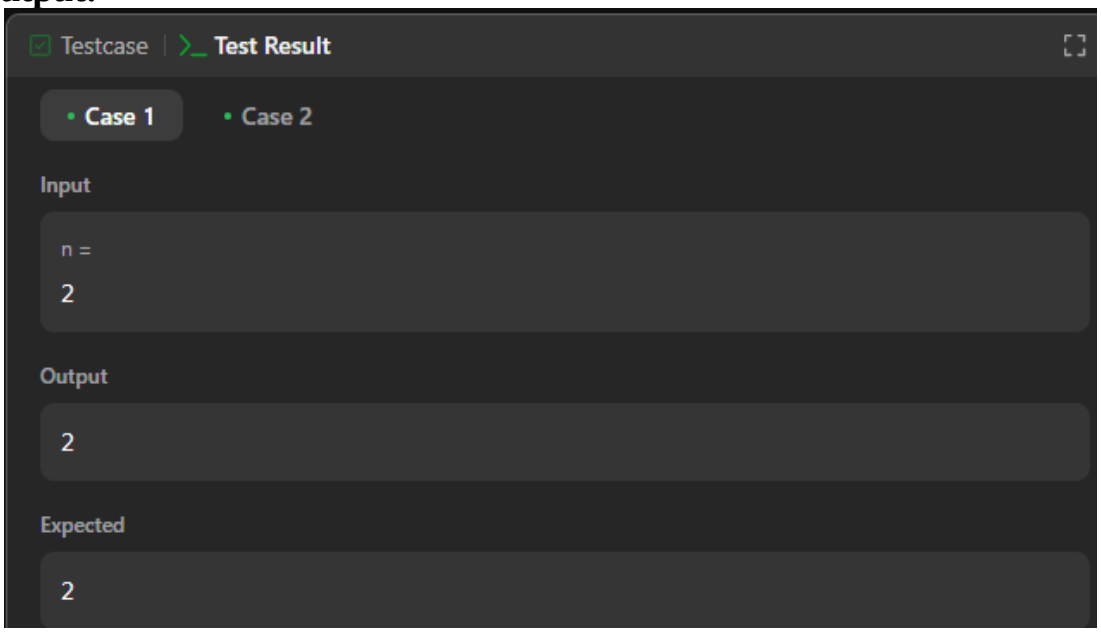
Problem 1. You are climbing a staircase. It takes n steps to reach the top.
Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top.

- **Code:**

```
class Solution {
public int climbStairs(int n) {
if (n == 1) return 1;
if (n == 2) return 2;
int first = 1, second = 2;

for (int i = 3; i <= n; i++) {
int third = first + second;
first = second;
second = third;
}return second;
}
}
```

- **Output:**



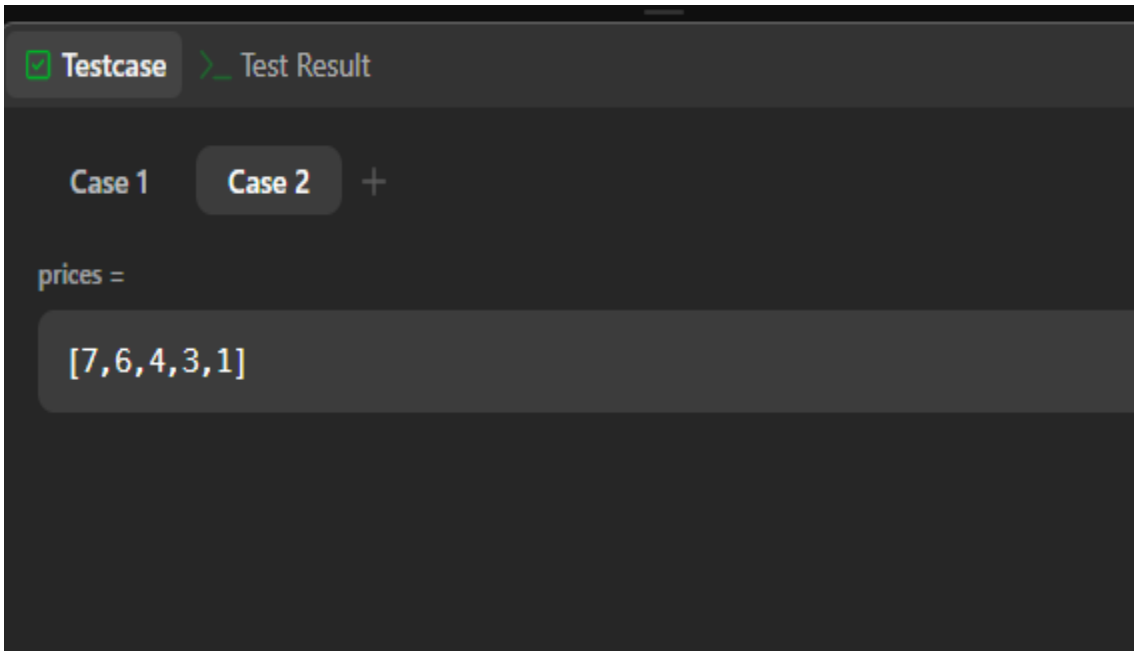
Problem 2. You are given an array prices where prices[i] is the stock price on the ith day Find the maximum profit you can achieve by buying and selling only once.

- **Code:**

```
class Solution {
public int maxProfit(int[] prices) {
int minPrice = Integer.MAX_VALUE;
int maxProfit = 0;

for (int price : prices) {
if (price < minPrice) {
minPrice = price;  // Update min buy price
} else {
maxProfit = Math.max(maxProfit, price - minPrice);  // Calculate max profit
}
}
return maxProfit;
}
}
```

- **Output:**



Problem 3. Given an integer array nums, find the contiguous subarray (containing at least one number) with the largest sum and return its sum.

- **Code:**

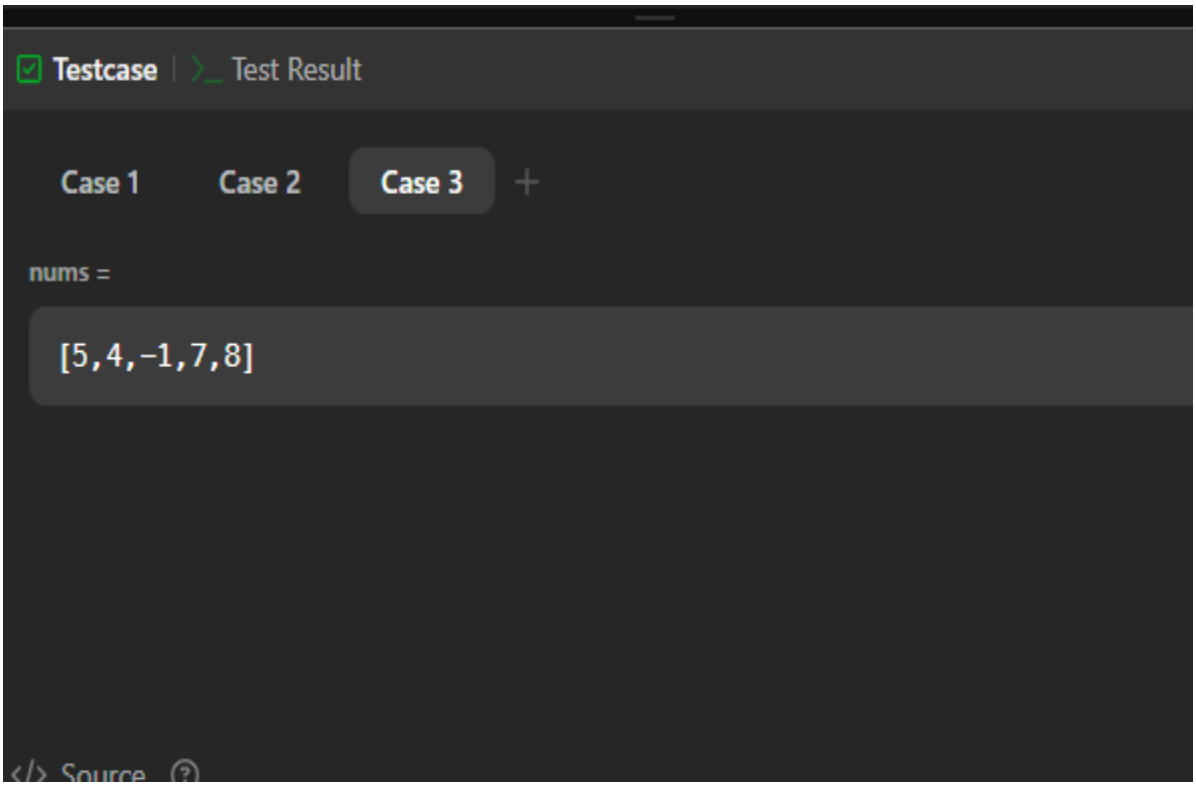
```
class Solution {
public int maxSubArray(int[] nums) {

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:**



Problem 4. You are a professional robber planning to rob houses along a street. Each house has money but adjacent houses cannot be robbed consecutively.Find the maximum money you can rob without alerting the police.

- **Code:**

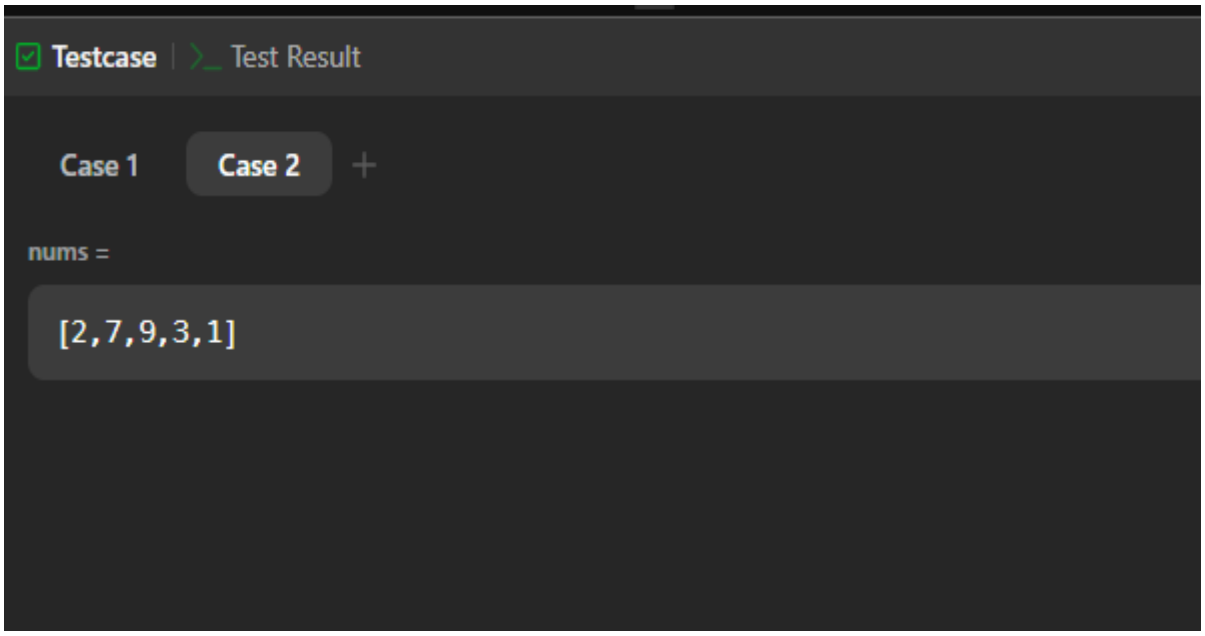
```
class Solution {
public int rob(int[] nums) {
if (nums.length == 0) return 0;
if (nums.length == 1) return nums[0];

int prev2 = 0;  // Maximum money robbed till i-2 house
int prev1 = nums[0];  // Maximum money robbed till i-1 house

for (int i = 1; i < nums.length; i++) {
int current = Math.max(prev1, prev2 + nums[i]);
prev2 = prev1;
prev1 = current;
}

return prev1;
}
}
```

- **Output:**

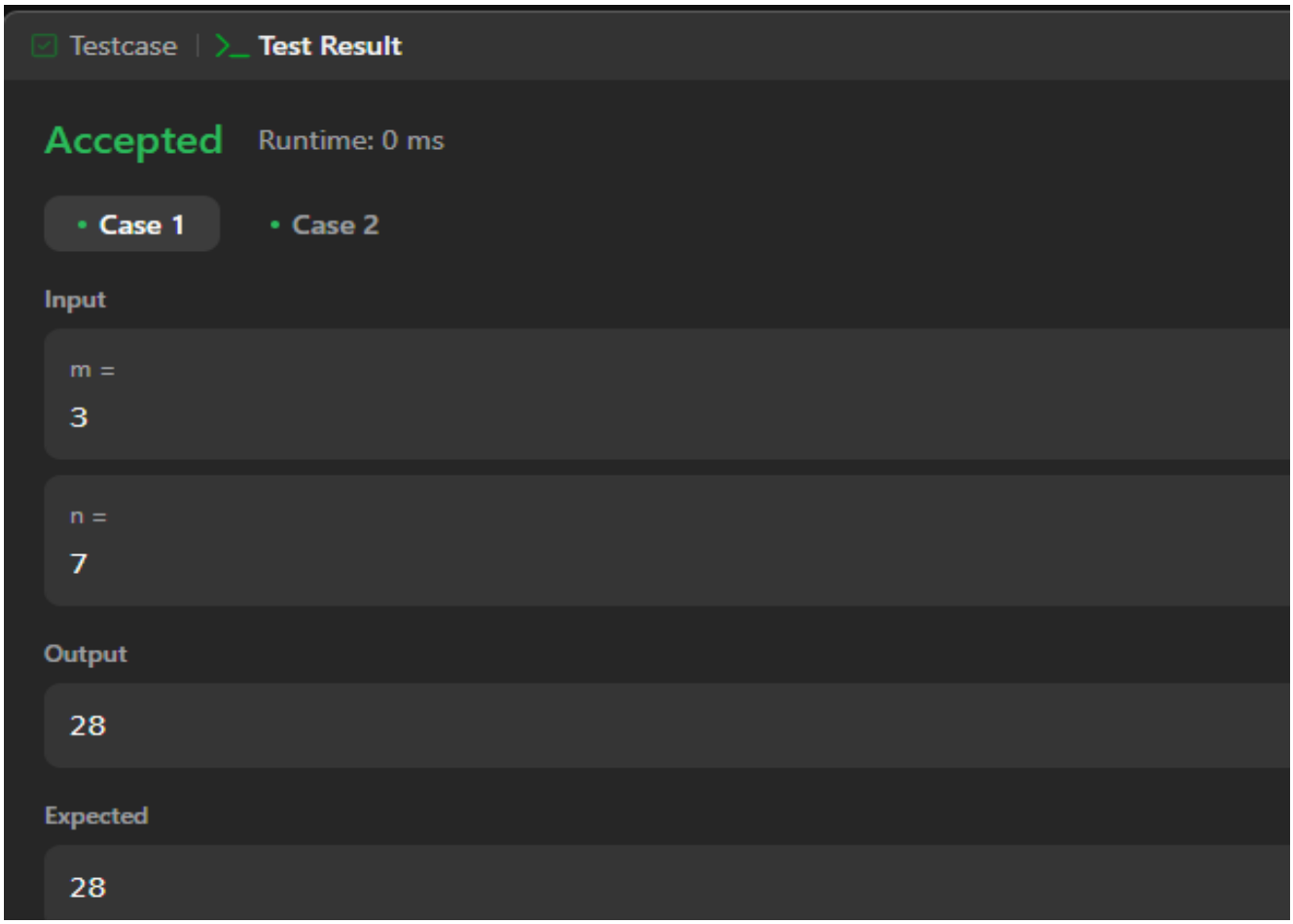


Problem 5. A robot moves in an m x n grid from the top-left to the bottom-right corner. The robot can only move right or down.Find the number of unique paths.

- **Code:**

```
class Solution {
public int uniquePaths(int m, int n) {
long result = 1;
for (int i = 1; i <= m - 1; i++) {
result = result * (n - 1 + i) / i;        // Computing with formula C(m+n-2, m-1)
}
return (int) result;
}
```

- **Output:**



}

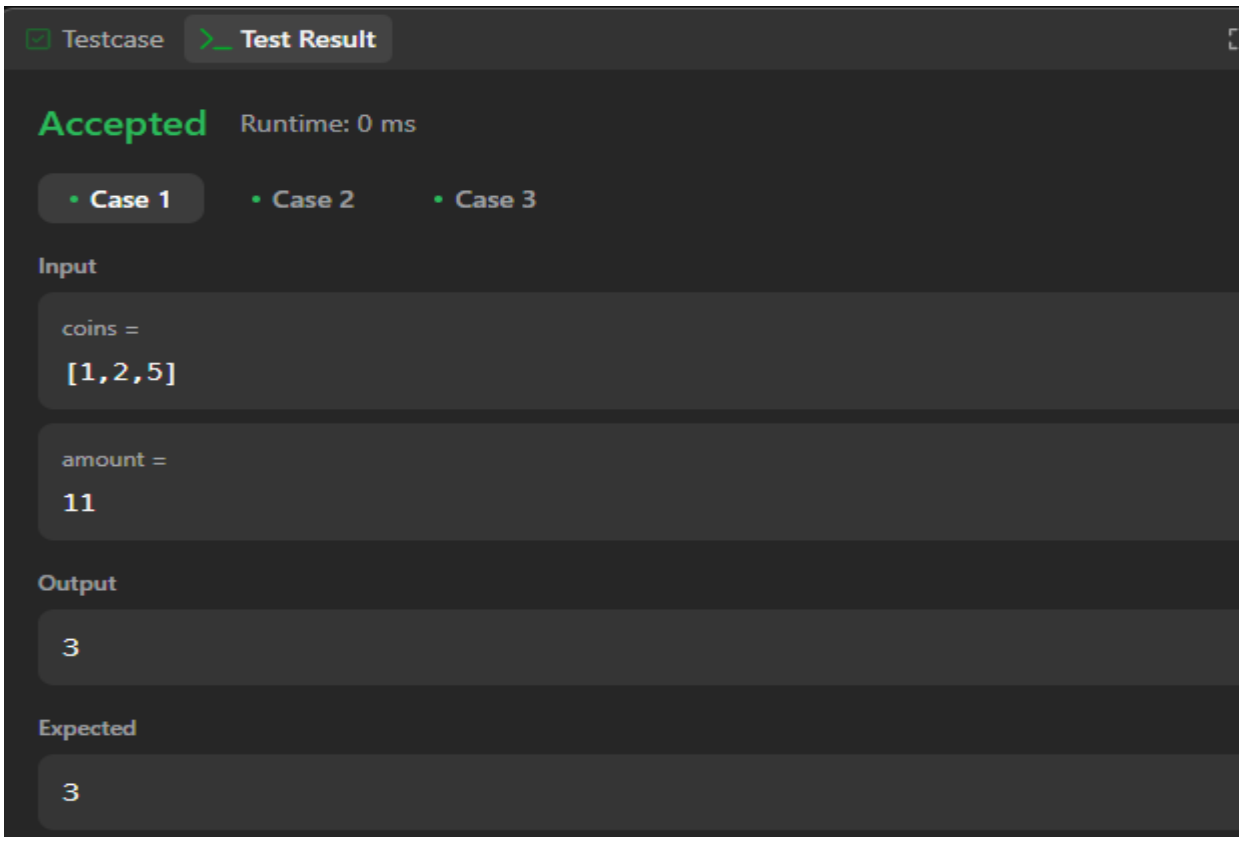
Problem 6. Given coins
of different denominations and an amount amount, return the fewest coins needed to make up that amount.

- **Code:**

```
class Solution {
public int coinChange(int[] coins, int amount) {
int max = amount + 1;
int[] dp = new int[amount + 1];
Arrays.fill(dp, max);
dp[0] = 0;  // Base case: 0 coins needed for amount 0

for (int coin : coins) {
for (int j = coin; j <= amount; j++) {
dp[j] = Math.min(dp[j], dp[j - coin] + 1);
}
}
return (dp[amount] == max) ? -1 : dp[amount];
}
}
```

- **Output:**



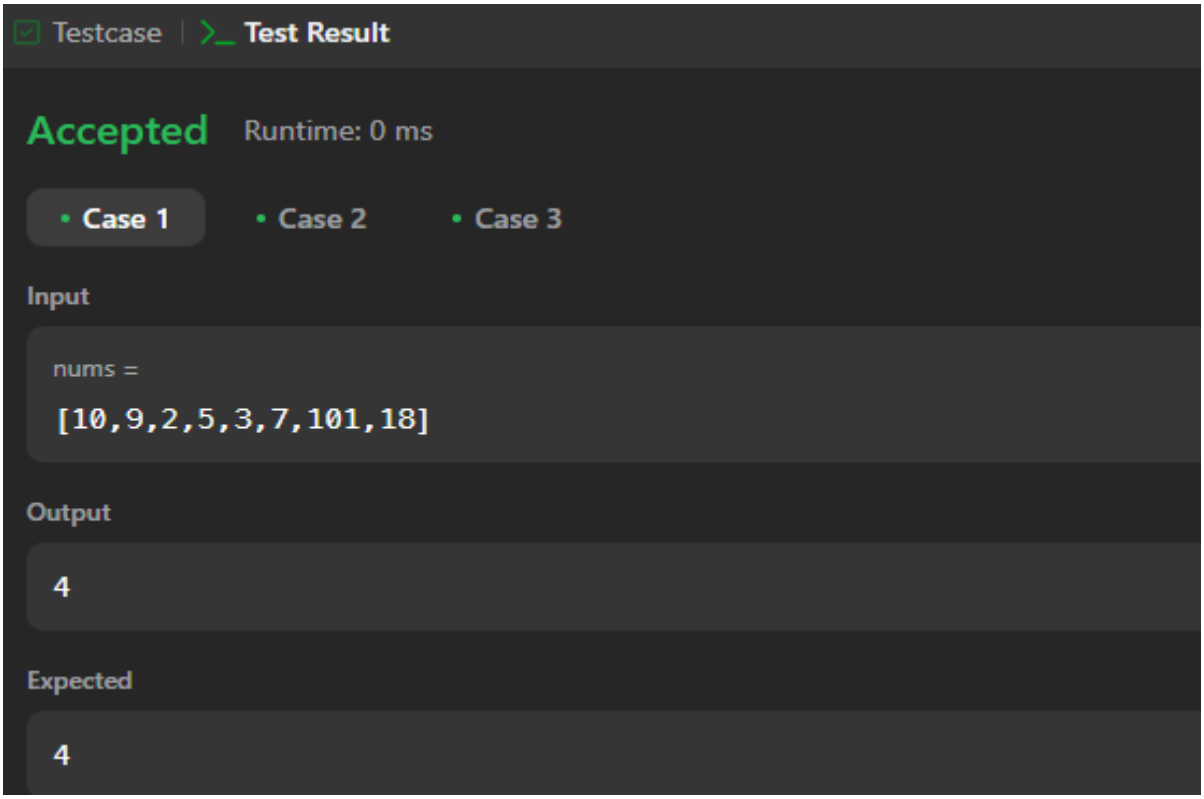
Problem 7. Given an array nums, return the length of the longest increasing subsequence.

- **Code:**

```
class Solution {
public int lengthOfLIS(int[] nums) {
int n = nums.length;
int[] dp = new int[n];
Arrays.fill(dp, 1); // Each element is at least length 1

int maxLength = 1;
for (int i = 1; i < n; i++) {
for (int j = 0; j < i; j++) {
if (nums[j] < nums[i]) {
dp[i] = Math.max(dp[i], dp[j] + 1);
}
}
maxLength = Math.max(maxLength, dp[i]);
}
return maxLength;
}  }
```

- **Output:**



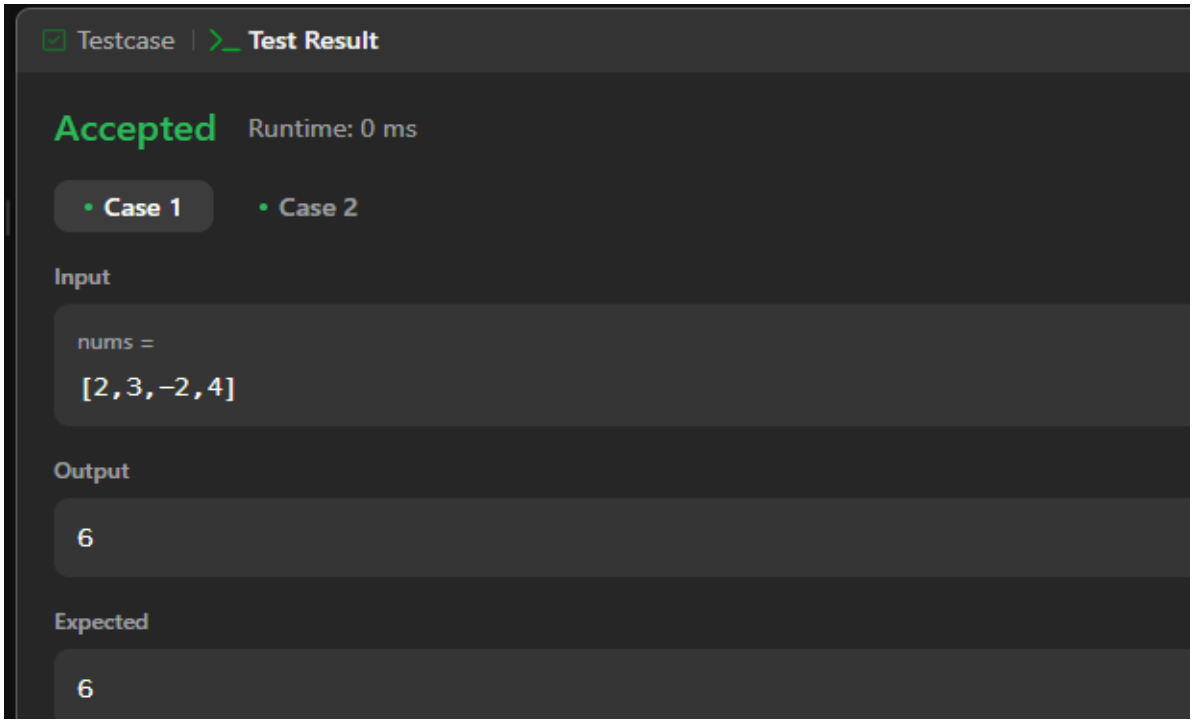
Problem 8. Find the contiguous subarray with the largest product.

- **Code:**

```
class Solution {
public int maxProduct(int[] nums) {
if (nums.length == 0) return 0;
int maxProduct = nums[0];
int currMax = nums[0];
int currMin = nums[0];

for (int i = 1; i < nums.length; i++) {
if (nums[i] < 0) {
int temp = currMax;
currMax = currMin;
currMin = temp;
}
currMax = Math.max(nums[i], currMax * nums[i]);
currMin = Math.min(nums[i], currMin * nums[i]);
maxProduct = Math.max(maxProduct, currMax);
}
return maxProduct;
}
}
```

- **Output:**

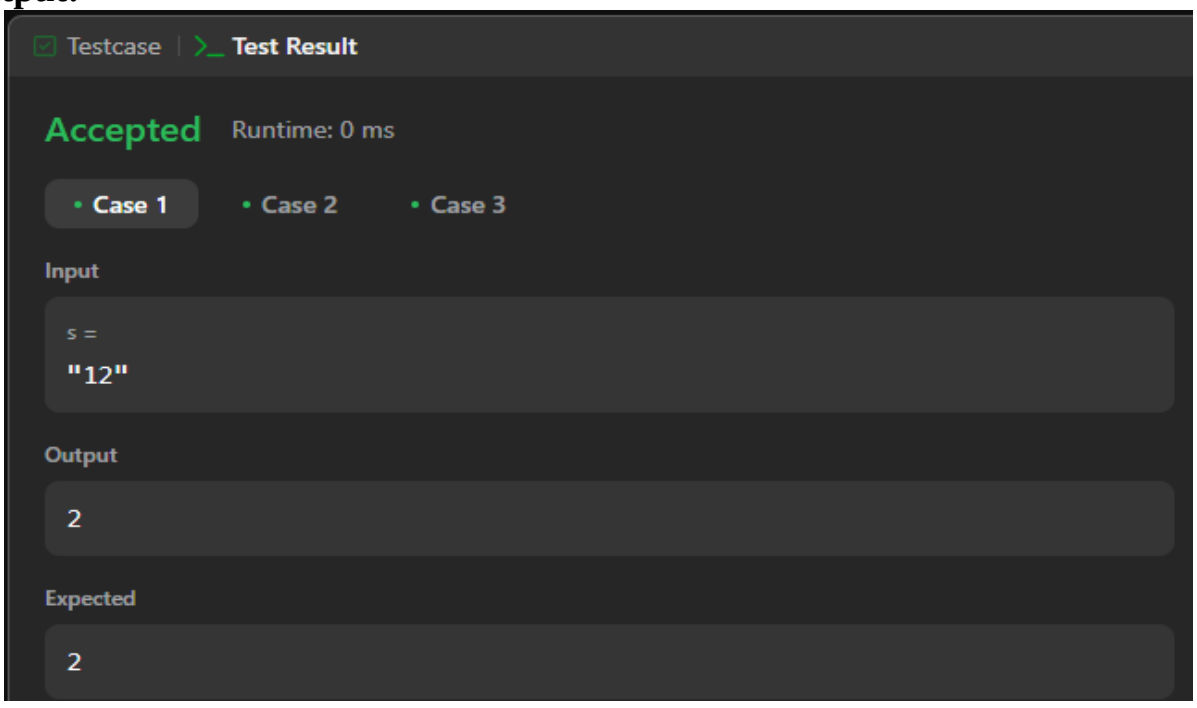


Problem 9. A string s containing digits maps to letters (A = 1, B = 2, ..., Z = 26). Return the number of ways to decode s.

- **Code:**

```
class Solution {
public int numDecodings(String s) {
if (s == null || s.length() == 0 || s.charAt(0) == '0') return 0;
int n = s.length();
int prev2 = 1;  // dp[i-2]
int prev1 = s.charAt(0) != '0' ? 1 : 0;
for (int i = 2; i <= n; i++) {
int curr = 0;
int oneDigit = Integer.parseInt(s.substring(i - 1, i));
int twoDigits = Integer.parseInt(s.substring(i - 2, i));
if (oneDigit >= 1) {
curr += prev1;
}
if (twoDigits >= 10 && twoDigits <= 26) {
curr += prev2;
}
prev2 = prev1;
prev1 = curr;
}
return prev1;
}   }
```

- **Output:**

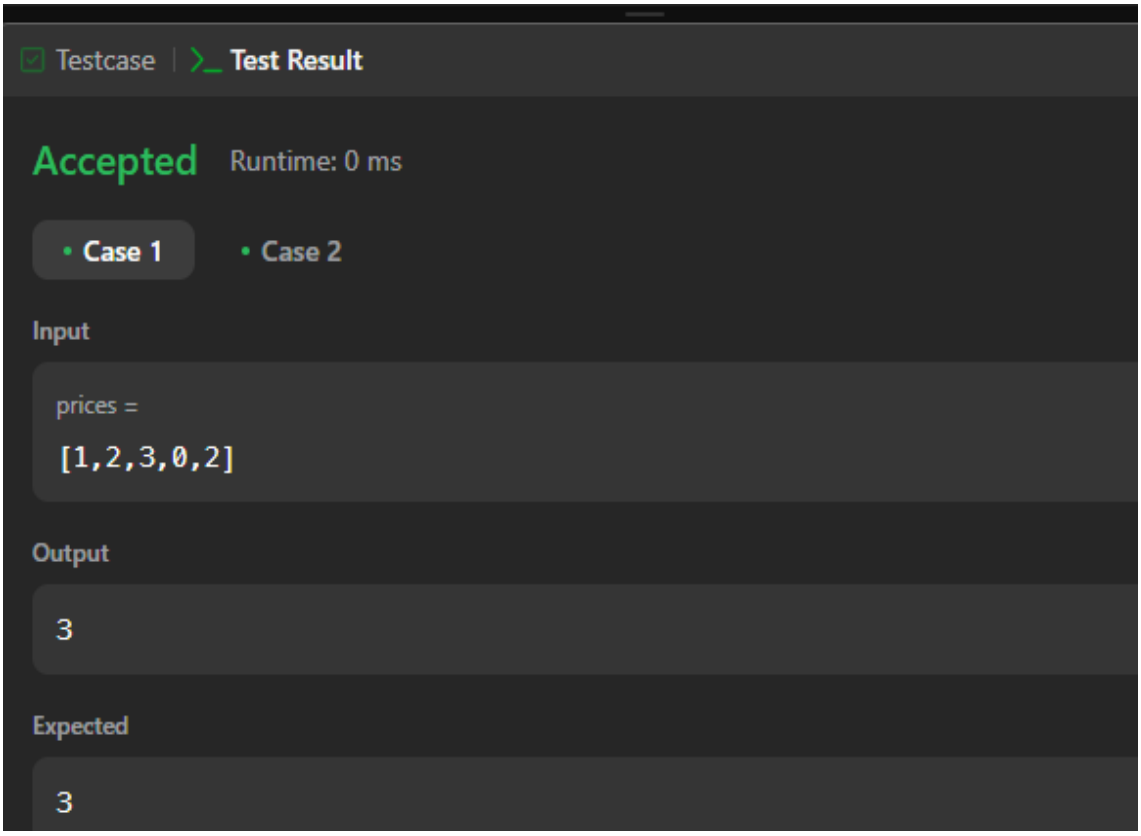


Problem 10: You may buy and sell stock, but after selling, you must wait 1 day before buying again.

- **Code:**

```
class Solution {
public int maxProfit(int[] prices) {
if (prices == null || prices.length == 0) return 0;
int buy = Integer.MIN_VALUE, sell = 0, cooldown = 0;
for (int price : prices) {
int prevSell = sell;
sell = buy + price;  // Sell today
buy = Math.max(buy, cooldown - price);  // Buy today
cooldown = Math.max(cooldown, prevSell);  // Cooldown from previous day
}
return Math.max(sell, cooldown); // Maximum profit is either in sell or cooldown state
}
}
```

- **Output:**

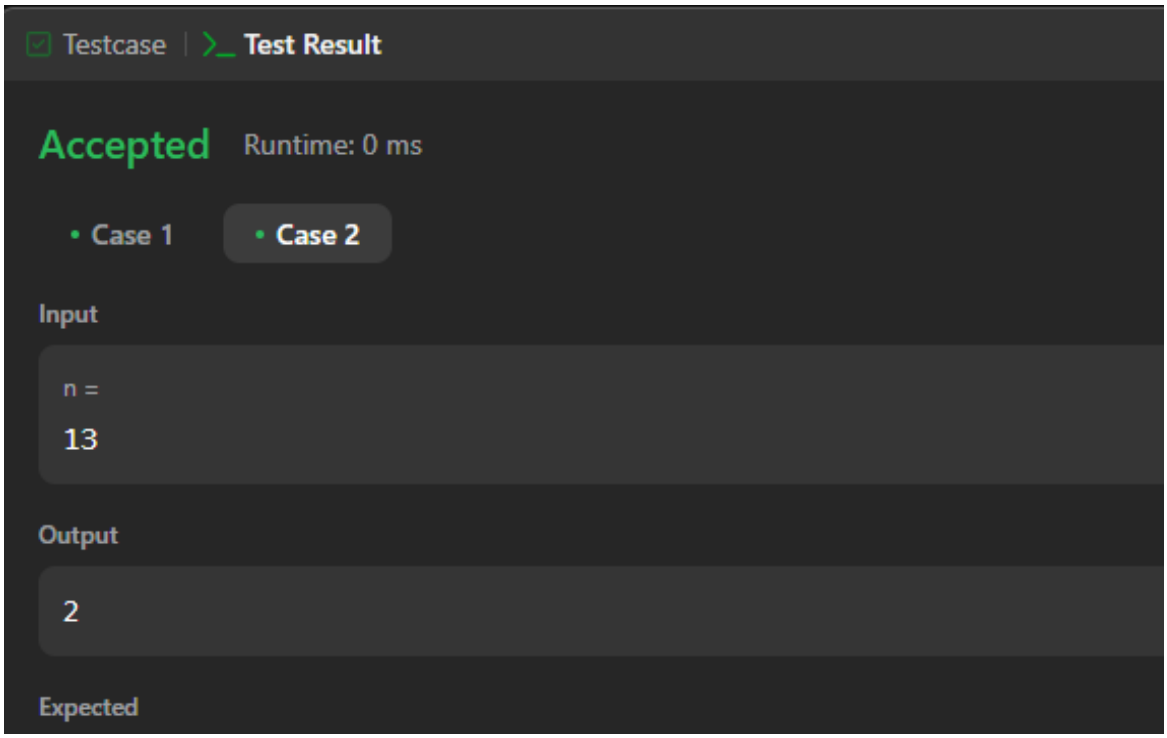


Problem 11. Given an integer n, return the fewest number of perfect squares that sum to n.

- **Code:**

```
class Solution {
public int numSquares(int n) {
if (isPerfectSquare(n)) return 1;
for (int i = 1; i * i <= n; i++) {
if (isPerfectSquare(n - i * i)) return 2;
}
while (n % 4 == 0) {
n /= 4;
}
if (n % 8 == 7) return 4;
  return 3;
}
private boolean isPerfectSquare(int x) {
int s = (int) Math.sqrt(x);
return s * s == x;
}
}
```

- **Output:**



Problem 12: Given a string s and a dictionary of strings wordDict, return true if s can be segmented into a space-separated sequence of one or more dictionary words.

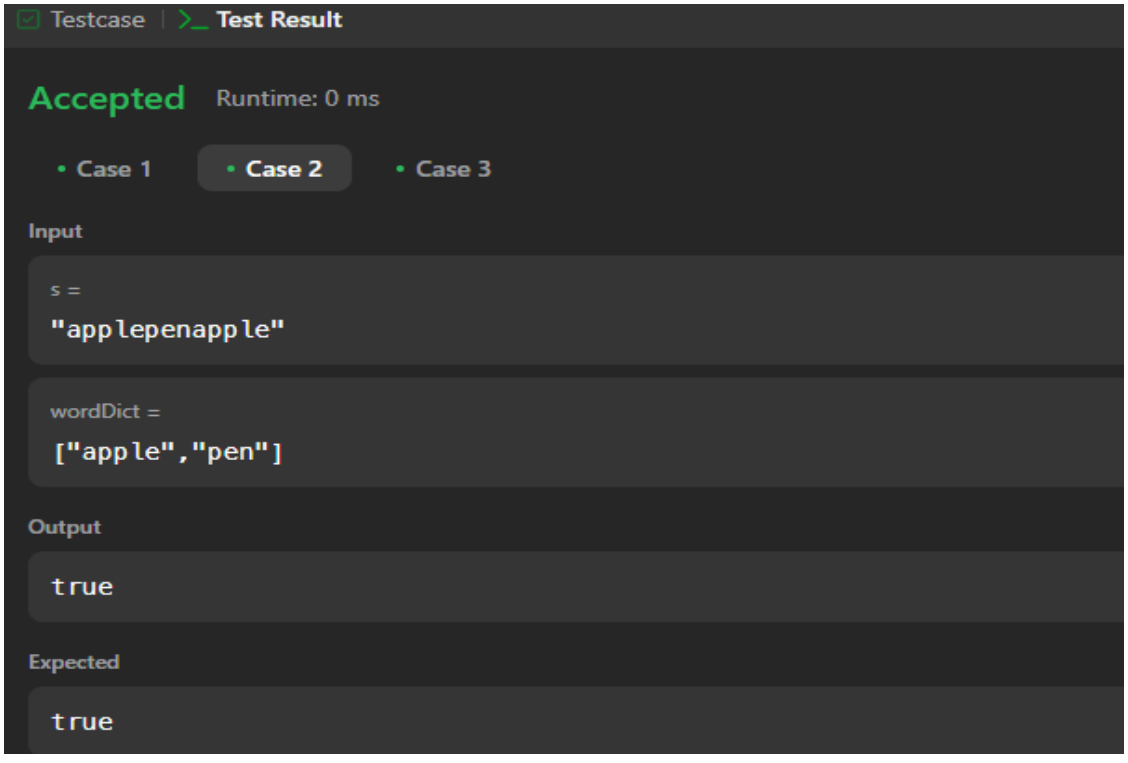
- **Code:**

```
class Solution {
public boolean wordBreak(String s, List<String> wordDict) {
Set<String> wordSet = new HashSet<>(wordDict);
Queue<Integer> queue = new LinkedList<>();
boolean[] visited = new boolean[s.length() + 1];

queue.add(0);
while (!queue.isEmpty()) {
int start = queue.poll();
if (start == s.length()) return true;

for (int end = start + 1; end <= s.length(); end++) {
if (!visited[end] && wordSet.contains(s.substring(start, end))) {
queue.add(end);
visited[end] = true;
} }}
return false;
}
}
```

- **Output:**



Problem 13. Given a string s and a dictionary of strings wordDict, add spaces in s to construct a sentence where each word is a valid dictionary word. Return all such possible sentences in any order.

- **Code:**

```
class Solution {
public List<String> wordBreak(String s, List<String> wordDict) {
Set<String> wordSet = new HashSet<>(wordDict);
Map<String, List<String>> memo = new HashMap<>();
return dfs(s, wordSet, memo);
}
private List<String> dfs(String s, Set<String> wordSet, Map<String, List<String>> memo) {
if (memo.containsKey(s)) return memo.get(s);
if (s.isEmpty()) return Arrays.asList("");
List<String> res = new ArrayList<>();
for (int i = 1; i <= s.length(); i++) {
String prefix = s.substring(0, i);
if (wordSet.contains(prefix)) {
List<String> suffixWays = dfs(s.substring(i), wordSet, memo);
for (String suffix : suffixWays) {
res.add(prefix + (suffix.isEmpty() ? "" : " ") + suffix);
}  } }
memo.put(s, res);
return res;
}     }
```

- **Output:**