Experiment-7

Student Name: Anushka **UID: 22BET10126**
Branch: BE-IT **Section/Group: 22BET_IOT-702/B**
Semester: 6th **Date of Performance: 20th Mar, 2025**
Subject Name: Advanced Programming Lab-2
Subject Code: 22ITP-351

Problem-1

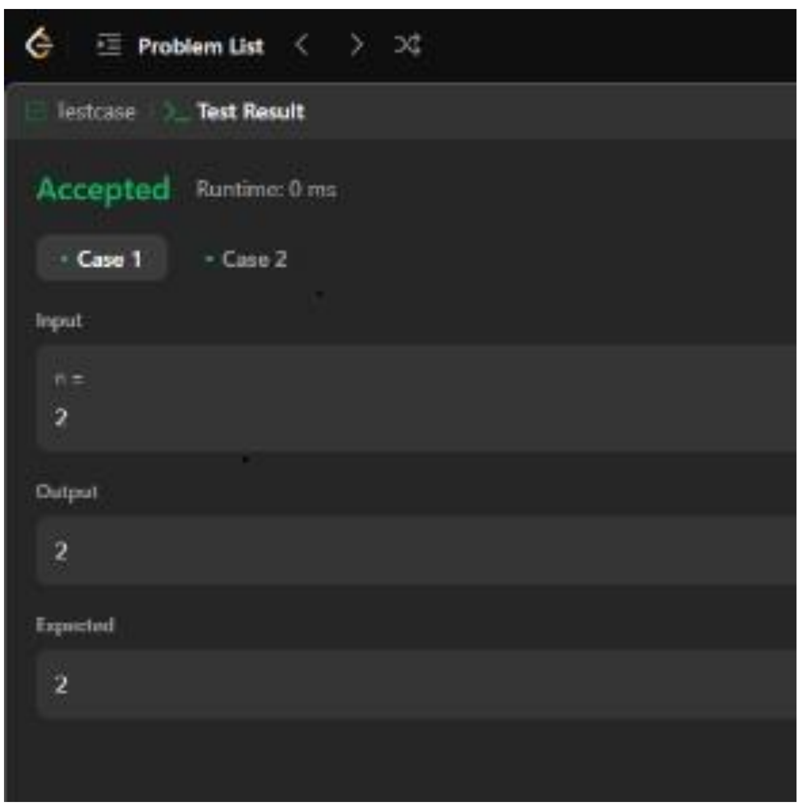
1. **Aim:** To develop a Java program that calculates the number of distinct ways to climb a staircase with n steps, where one can take either 1 step or 2 steps at a time.

2. **Code:**

```
import java.util.Scanner;

class Solution { // Change class name to Solution
    public int climbStairs(int n) {
        if (n <= 2) return n; // Base cases

        int prev1 = 1, prev2 = 2, current = 0;

        for (int i = 3; i <= n; i++) {
            current = prev1 + prev2; // Fibonacci logic
            prev1 = prev2;
            prev2 = current;
        }

        return current;
    }
}

public class Main { // Separate class for input handling
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter number of steps (1 to 45): ");
```

```
        int n = 0;
        while (true) {
            if (sc.hasNextInt()) {
                n = sc.nextInt();
                if (n >= 1 && n <= 45) break; // Valid range
                else System.out.print("Invalid input! Enter a number between 1 and 45: ");
            } else {
                System.out.print("Invalid input! Please enter a valid integer: ");
                sc.next(); // Consume invalid input
            }
        }

        Solution sol = new Solution(); // Create an object of Solution
        System.out.println("Total distinct ways to climb: " + sol.climbStairs(n));
        sc.close();
    }
}
```

3. Output:



Problem-2

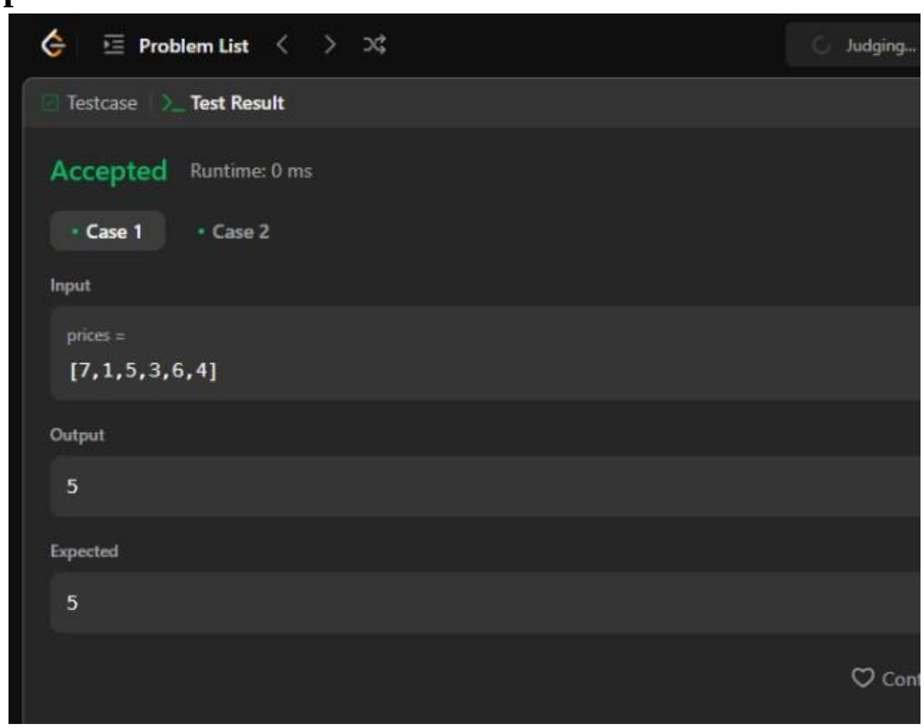
1. **Aim:** To develop a Java program that finds the **maximum profit** that can be obtained from a given array of stock prices, where you can buy and sell once.

2. **Code:**

```
import java.util.Scanner;

class Solution { // Class name changed to "Solution"
    public int maxProfit(int[] prices) {
        if (prices == null || prices.length < 2) return 0;

        int minPrice = prices[0];
        int maxProfit = 0;

        for (int i = 1; i < prices.length; i++) {
            if (prices[i] < minPrice) {
                minPrice = prices[i]; // Update minimum price
            } else {
                maxProfit = Math.max(maxProfit, prices[i] - minPrice); // Calculate max profit
            }
        }
        return maxProfit;
    }
}

public class Main { // Separate class for handling input
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter number of days: ");
        int n = sc.nextInt();

        if (n < 1 || n > 100000) {
            System.out.println("Invalid input! n should be between 1 and 100000.");
            return;
        }
```

```
        int[] prices = new int[n];
        System.out.println("Enter stock prices: ");
        for (int i = 0; i < n; i++) {
            prices[i] = sc.nextInt();
            if (prices[i] < 0 || prices[i] > 10000) {
                System.out.println("Invalid price! Prices should be between 0 and 10000.");
                return;
            }
        }

        Solution sol = new Solution(); // Create an object of Solution
        System.out.println("Maximum Profit: " + sol.maxProfit(prices));
        sc.close();
    }
}
```

3. Output:

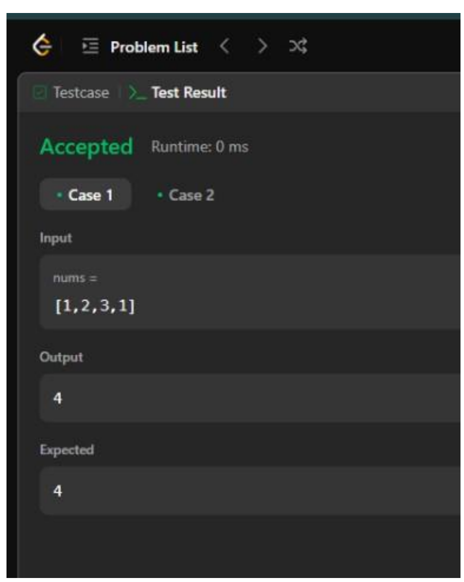


Problem-3

1. **Aim:** To develop a Java program that determines the **maximum amount of money** a robber can steal without robbing adjacent houses.
2. **Code:**

```
import java.util.Arrays;
class Solution {
   private int[] dp;
   public int rob(int[] nums) {
      dp = new int[nums.length];
      Arrays.fill(dp, -1); // Initialize memoization array
      return robHelper(nums, nums.length - 1);
   }
   private int robHelper(int[] nums, int i) {
      if (i < 0) return 0; // Base case
      if (dp[i] != -1) return dp[i]; // Return stored result
      // Either rob the current house and skip the next or skip this house
      dp[i] = Math.max(robHelper(nums, i - 1), nums[i] + robHelper(nums, i - 2));
      return dp[i];
   }
}
```

3. **Output:**



Problem-4

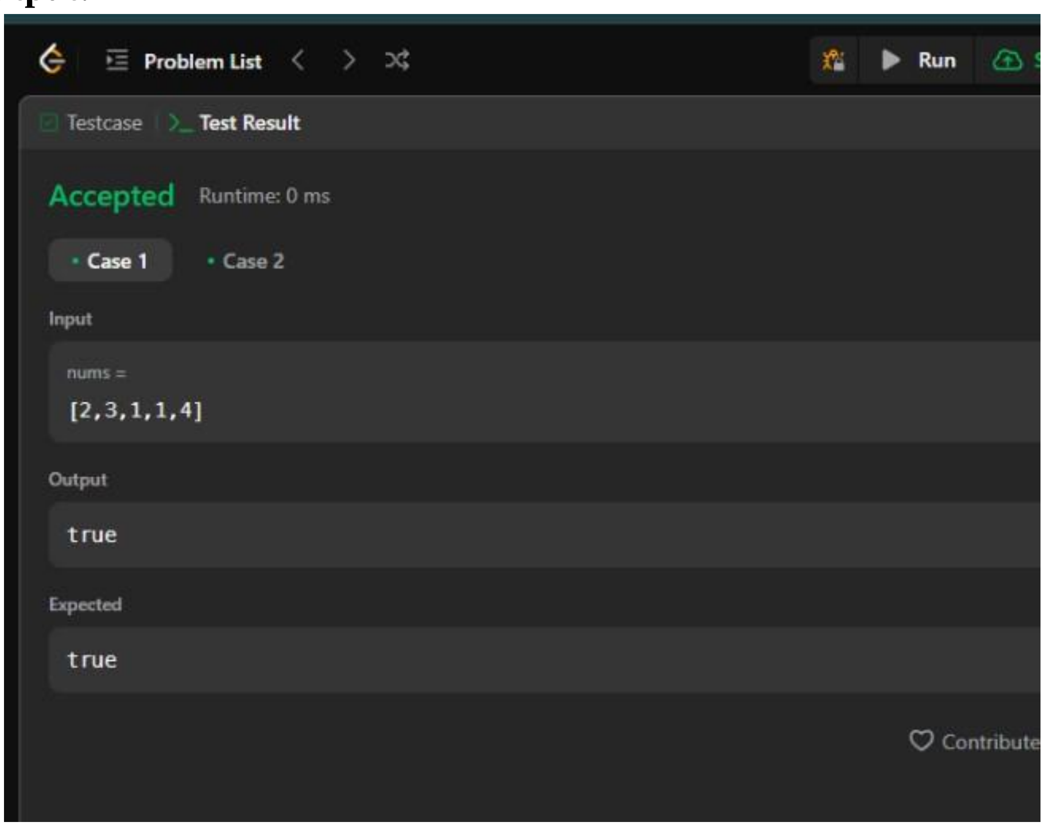
1. **Aim:** To implement a Java program that determines whether you can reach the last index of an array, given that each element represents the maximum number of steps you can jump forward.

2. **Code:**

```
import java.util.Scanner{
class Solution {
    public boolean canJump(int[] nums) {
        int maxReach = 0; // Track the farthest index we can reach
        for (int i = 0; i < nums.length; i++) {
            if (i > maxReach) return false; // If current index is unreachable
            maxReach = Math.max(maxReach, i + nums[i]); // Update max reach
            if (maxReach >= nums.length - 1) return true; // Check if we can reach last index
        }
        return false;
    }
}public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter number of elements: ");
        int n = sc.nextInt();

        if (n < 1 || n > 10000) {
            System.out.println("Invalid input! Length should be between 1 and 10^4.");
            return;
        }
        int[] nums = new int[n];
```

```java
        System.out.println("Enter the elements: ");
        for (int i = 0; i < n; i++) {
            nums[i] = sc.nextInt();
            if (nums[i] < 0 || nums[i] > 100000) {
                System.out.println("Invalid input! Values should be between 0 and 10^5.");
                return;
            }
        }
        Solution sol = new Solution();
        System.out.println("Can reach last index: " + sol.canJump(nums));
        sc.close();
    }
}
```

3. Output:



Problem-5

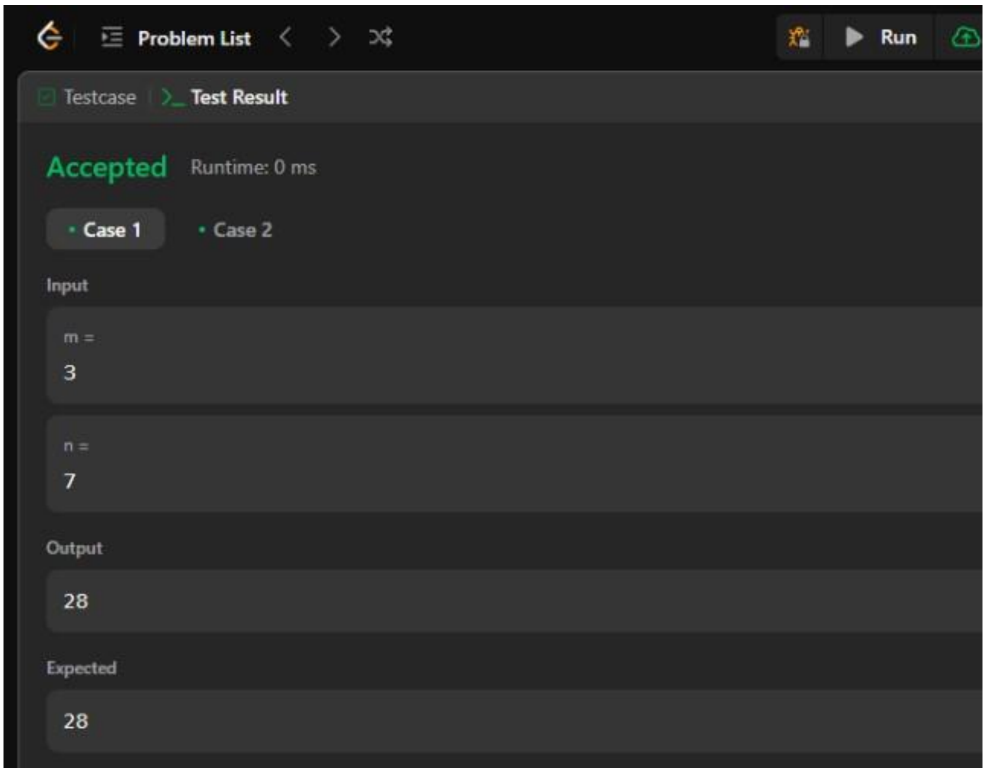
1. **Aim:** To develop a Java program that computes the number of **unique paths** a robot can take in an **m × n** grid while only moving right or down.

2. **Code:**

```
import java.util.Scanner;

class Solution {
    public int uniquePaths(int m, int n) {
        int[][] dp = new int[m][n];

        // Initialize first row and first column to 1
        for (int i = 0; i < m; i++) dp[i][0] = 1;
        for (int j = 0; j < n; j++) dp[0][j] = 1;

        // Fill DP table
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
            }
        }
        return dp[m - 1][n - 1]; // Bottom-right cell
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter grid size (m n): ");
        int m = sc.nextInt();
        int n = sc.nextInt();

        if (m <= 0 || n <= 0) {
            System.out.println("Invalid input! Grid dimensions must be positive.");
```

```
            return;
        }

        Solution sol = new Solution();
        System.out.println("Unique Paths: " + sol.uniquePaths(m, n));
        sc.close();
    }
}
```

3. **Output:**



Problem-6

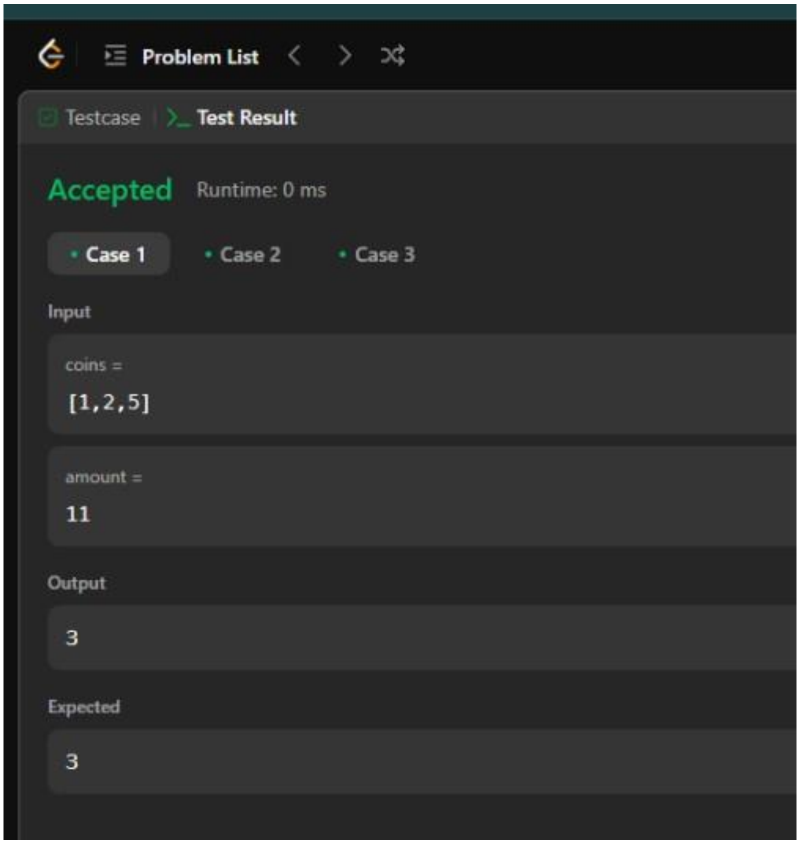
1. **Aim:** To develop a **Java program** that computes the **minimum number of coins** needed to form a given amount using a **dynamic programming approach**.
2. **Code:**

```
import java.util.Arrays;
import java.util.Scanner;
```

```
class Solution {
    public int coinChange(int[] coins, int amount) {
        int max = amount + 1; // Set an unreachable value
        int[] dp = new int[amount + 1];
        Arrays.fill(dp, max); // Fill with max value
        dp[0] = 0; // Base case

        // Iterate over all amounts
        for (int i = 1; i <= amount; i++) {
            for (int coin : coins) {
                if (i >= coin) { // Check if coin can be used
                    dp[i] = Math.min(dp[i], 1 + dp[i - coin]);
                }
            }
        }
        return (dp[amount] == max) ? -1 : dp[amount]; // If unreachable, return -1
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // Taking input dynamically
        System.out.print("Enter number of coin types: ");
        int n = sc.nextInt();
        int[] coins = new int[n];

        System.out.print("Enter coin denominations: ");
        for (int i = 0; i < n; i++) {
            coins[i] = sc.nextInt();
        }

        System.out.print("Enter amount: ");
        int amount = sc.nextInt();

        Solution sol = new Solution();
        int result = sol.coinChange(coins, amount);

        System.out.println("Minimum coins required: " + result);
        sc.close();
```

```
    }
}
```

3. Output:



Problem-7

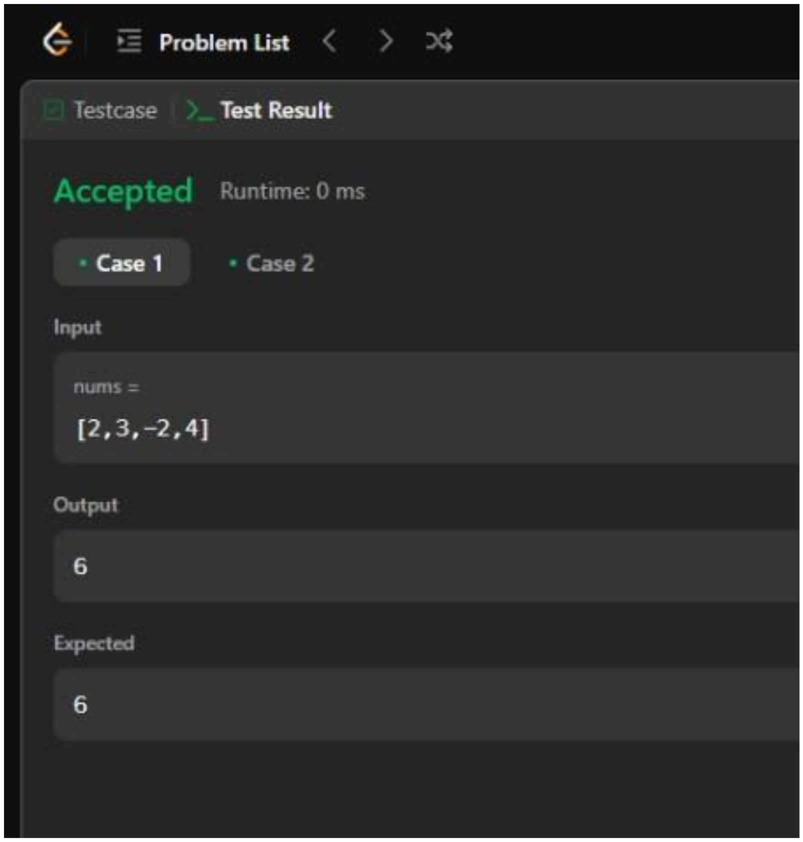
1. **Aim:** To implement a **Java program** that efficiently finds the **maximum product subarray** using **Dynamic Programming (DP) or Kadane's Algorithm variation**.
2. **Code:**

```
import java.util.Scanner;

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

```
        int maxProd = nums[0], minProd = nums[0], result = nums[0];

        for (int i = 1; i < nums.length; i++) {
            if (nums[i] < 0) { // Swap max and min when encountering a negative number
                int temp = maxProd;
                maxProd = minProd;
                minProd = temp;
            }

            maxProd = Math.max(nums[i], maxProd * nums[i]);
            minProd = Math.min(nums[i], minProd * nums[i]);

            result = Math.max(result, maxProd);
        }
        return result;
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // Taking input dynamically
        System.out.print("Enter array size: ");
        int n = sc.nextInt();
        int[] nums = new int[n];

        System.out.print("Enter array elements: ");
        for (int i = 0; i < n; i++) {
            nums[i] = sc.nextInt();
        }

        Solution sol = new Solution();
        int result = sol.maxProduct(nums);

        System.out.println("Maximum product subarray: " + result);
        sc.close();
    }
```

```
}
```

3. Output:



Problem-8

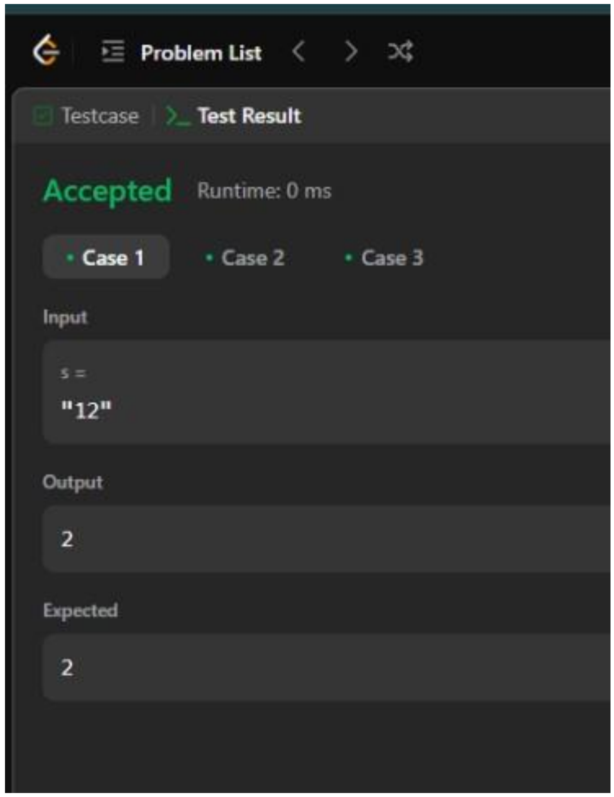
1. **Aim:** To implement a **Java program** that finds the number of ways to decode a given numeric string using **Dynamic Programming (DP).**

2. **Code:**

```
import java.util.Scanner;

class Solution {
    public int numDecodings(String s) {
        if (s == null || s.length() == 0 || s.charAt(0) == '0') return 0;
```

```
        int n = s.length();
        int[] dp = new int[n + 1];
        dp[0] = 1; // Empty string has 1 way to decode
        dp[1] = s.charAt(0) != '0' ? 1 : 0; // Single character decoding

        for (int i = 2; i <= n; i++) {
            int oneDigit = Integer.parseInt(s.substring(i - 1, i)); // Last 1 digit
            int twoDigits = Integer.parseInt(s.substring(i - 2, i)); // Last 2 digits

            if (oneDigit >= 1) dp[i] += dp[i - 1]; // If valid single character
            if (twoDigits >= 10 && twoDigits <= 26) dp[i] += dp[i - 2]; // If valid double character
        }

        return dp[n];
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter the encoded string: ");
        String s = sc.next();

        Solution sol = new Solution();
        int result = sol.numDecodings(s);

        System.out.println("Number of ways to decode: " + result);
        sc.close();
    }
}
```

3. Output:



Problem-9

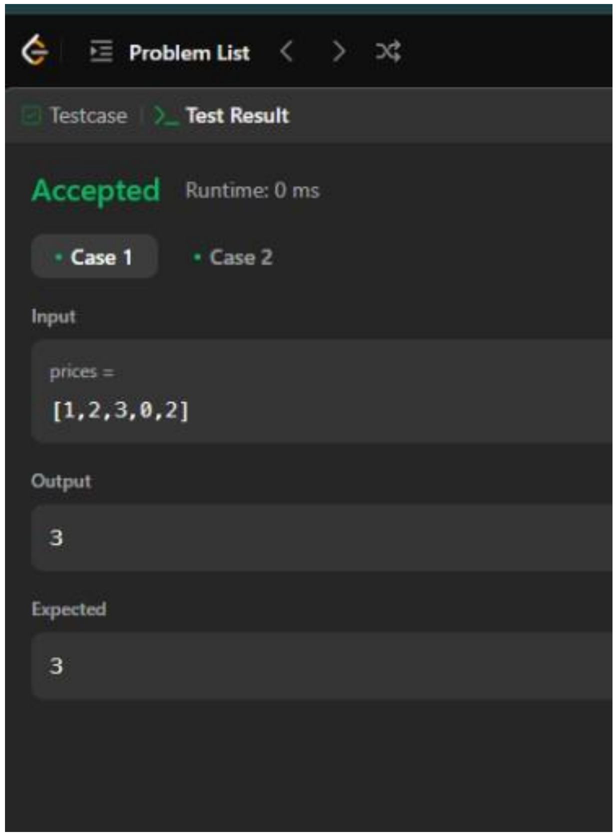
1. **Aim:** To implement a **Java program** that calculates the maximum possible profit by using **Dynamic Programming (DP).**

2. **Code:**

```
import java.util.Scanner;

class Solution {
    public int maxProfit(int[] prices) {
        if (prices == null || prices.length == 0) return 0;

        int n = prices.length;
        int[][] dp = new int[n][3];

        dp[0][0] = -prices[0]; // Buying on day 0
```

```
        dp[0][1] = 0; // Selling is not possible on day 0
        dp[0][2] = 0; // Cooldown is also 0 initially

        for (int i = 1; i < n; i++) {
            dp[i][0] = Math.max(dp[i - 1][0], dp[i - 1][2] - prices[i]); // Buy or continue holding
            dp[i][1] = dp[i - 1][0] + prices[i]; // Sell and move to cooldown
            dp[i][2] = Math.max(dp[i - 1][1], dp[i - 1][2]); // Continue cooldown
        }

        return Math.max(dp[n - 1][1], dp[n - 1][2]); // Max profit after last day
    }
}


public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter number of days: ");
        int n = sc.nextInt();

        int[] prices = new int[n];
        System.out.println("Enter stock prices:");
        for (int i = 0; i < n; i++) {
            prices[i] = sc.nextInt();
        }

        Solution sol = new Solution();

        int result = sol.maxProfit(prices);

        System.out.println("Maximum profit: " + result);
        sc.close();
    }
}
```

3. Output:



Problem-10

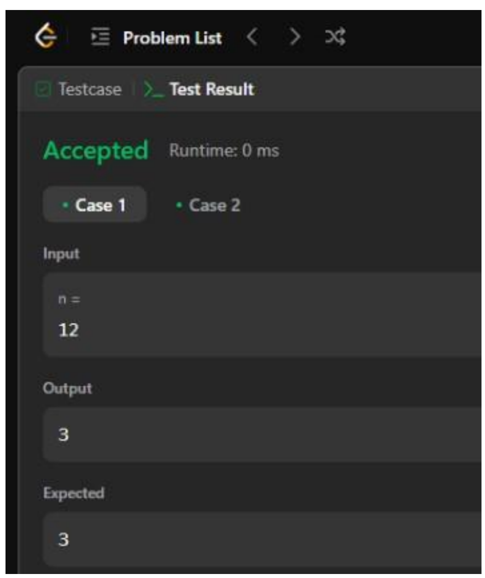
1. **Aim:** To implement an **efficient** algorithm in Java that finds the **minimum number of perfect squares** needed to sum up to n.

2. **Code:**

```
import java.util.Scanner;
import java.util.Arrays;

class Solution {
    public int numSquares(int n) {
        int[] dp = new int[n + 1];
        Arrays.fill(dp, Integer.MAX_VALUE);

        dp[0] = 0; // Base case
```

```
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j * j <= i; j++) {
                dp[i] = Math.min(dp[i], dp[i - j * j] + 1);
            }
        }
        return dp[n];
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter n: ");
        int n = sc.nextInt();
        Solution sol = new Solution();
        int result = sol.numSquares(n);

        System.out.println("Minimum number of perfect squares: " + result);
        sc.close();
    }
}
```

3. Output:



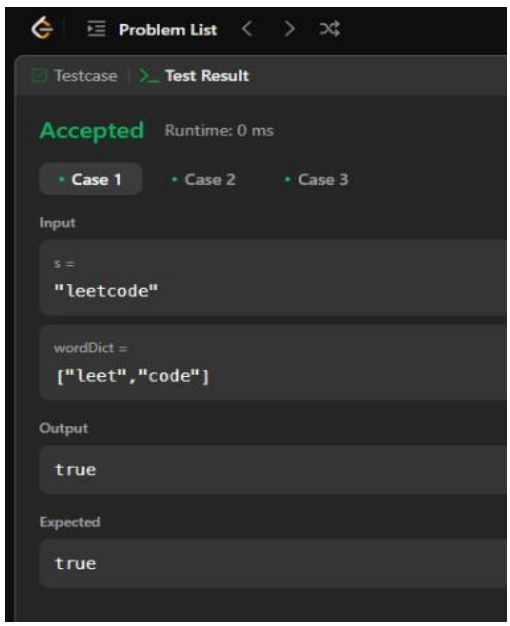
Problem-11

1. **Aim:** You are given a string s and a **dictionary of words** wordDict. Your task is to determine whether s can be **segmented** into a space-separated sequence of **one or more dictionary words**.
2. **Code:**

```
import java.util.*;

class Solution {
    public boolean wordBreak(String s, List<String> wordDict) {
        Set<String> wordSet = new HashSet<>(wordDict);
        int n = s.length();
        boolean[] dp = new boolean[n + 1];
        dp[0] = true;  // Base case (empty string)

        for (int i = 1; i <= n; i++) {
            for (int j = 0; j < i; j++) {
                if (dp[j] && wordSet.contains(s.substring(j, i))) {
                    dp[i] = true;
                    break;
                }
            }
        }
        return dp[n];
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter the string s: ");
        String s = sc.nextLine();

        System.out.print("Enter the number of words in the dictionary: ");
        int n = sc.nextInt();
        sc.nextLine();  // Consume the newline

        List<String> wordDict = new ArrayList<>();
```

```
            System.out.println("Enter dictionary words:");
            for (int i = 0; i < n; i++) {
                wordDict.add(sc.nextLine());
            }

            Solution sol = new Solution();
            boolean result = sol.wordBreak(s, wordDict);

            System.out.println("Can be segmented: " + result);
            sc.close();
        }
    }
```

3. Output:



Problem-12

1. **Aim:** The goal of this problem is to segment a given string s into valid sentences where each word appears in the given dictionary wordDict

2. **Code:**

```
import java.util.*;
```

```
public class Solution {

    public List<String> wordBreak(String s, List<String> wordDict) {
        Set<String> wordSet = new HashSet<>(wordDict); // Convert wordDict to a Set for fast
lookup
        Map<String, List<String>> memo = new HashMap<>(); // Memoization map
        return backtrack(s, wordSet, memo);
    }

    private List<String> backtrack(String s, Set<String> wordSet, Map<String, List<String>>
memo) {
        if (memo.containsKey(s)) {
            return memo.get(s); // Return cached results if already computed
        }

        List<String> result = new ArrayList<>();
        if (s.isEmpty()) {
            result.add(""); // Base case: return an empty string to form sentences
            return result;
        }

        // Iterate through all possible substrings
        for (int i = 1; i <= s.length(); i++) {
            String prefix = s.substring(0, i);
            if (wordSet.contains(prefix)) {
                List<String> suffixWays = backtrack(s.substring(i), wordSet, memo);
                for (String suffix : suffixWays) {
                    result.add(prefix + (suffix.isEmpty() ? "" : " ") + suffix);
                }
            }
        }

        memo.put(s, result); // Cache the result for this substring
        return result;
    }

    public static void main(String[] args) {
        String s = "catsanddog";
        List<String> wordDict = Arrays.asList("cat", "cats", "and", "sand", "dog");

        Solution solution = new Solution();
```

```
        List<String> sentences = solution.wordBreak(s, wordDict);
        System.out.println(sentences);
    }
}
```

3. Output: