Experiment 7

Student Name: Rahul Prasad **UID:** 22BET10167
Branch: IT **Section/Group:** 701/A
Semester: 6th **Date :** 21/03/2025
Subject Name: Advanced Programming Lab - 2 **Subject Code:** 22ITP-351

1. <u>Problem 1:</u>

➢ **Unique Paths:**

There is a robot on an m x n grid. The robot is initially located at the top-left corner (i.e., grid[0][0]). The robot tries to move to the bottom-right corner (i.e., grid[m - 1][n - 1]). The robot can only move either down or right at any point in time.

Given the two integers m and n, return the number of possible unique paths that the robot can take to reach the bottom-right corner.

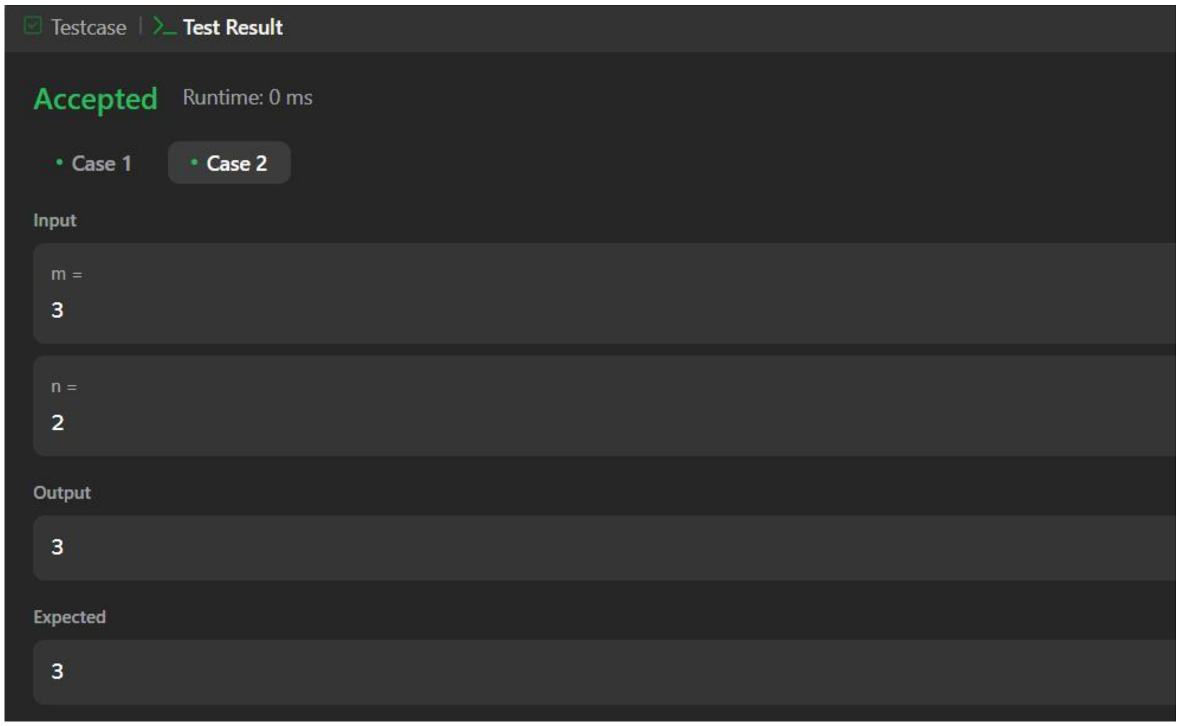
The test cases are generated so that the answer will be less than or equal to 2 * 109.

➢ **Code:**

```
class Solution {

    public int uniquePaths(int m, int n) {

        int[][] dp = new int[m][n];



        for (int i = 0; i < m; i++) dp[i][0] = 1;

        for (int j = 0; j < n; j++) dp[0][j] = 1;
```

```
                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];
            }
        }
```

➢ **Output**



Testcase | Test Result

Accepted Runtime: 0 ms

Case 1 Case 2

Input

```
m =
3
n =
2
```

Output

```
3
```

Expected

```
3
```

2. **<u>Problem 2:</u>**

- **Coin Change:**

You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money.

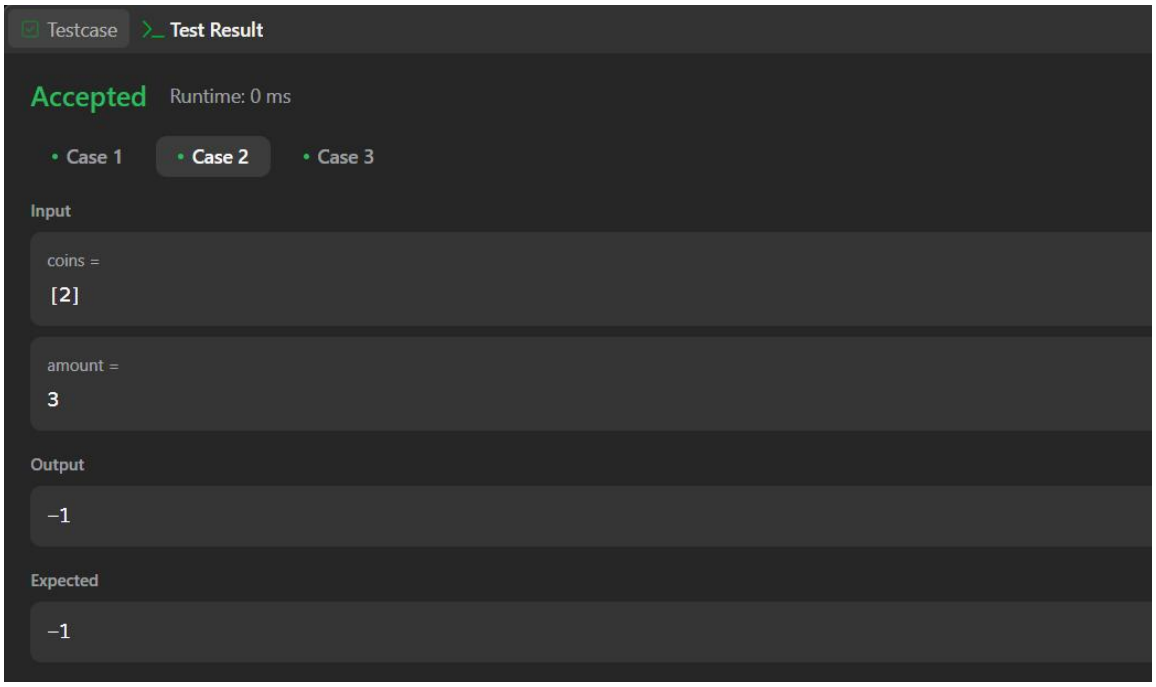
Return the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return -1.

You may assume that you have an infinite number of each kind of coin.

- **Code:**

```
class Solution {

   public int coinChange(int[] coins, int amount) {

      int[] dp = new int[amount + 1];

      Arrays.fill(dp, amount + 1);

      dp[0] = 0;



      for (int coin : coins) {

         for (int i = coin; i <= amount; i++) {

            dp[i] = Math.min(dp[i], dp[i - coin] + 1);

         }

      }

      return dp[amount] == amount + 1 ? -1 : dp[amount];

   }

}
```

- **Output:**



```
Accepted    Runtime: 0 ms

Input
coins =
[2]
amount =
3
Output
-1
Expected
-1
```

3. <u>Problem - 3:</u>

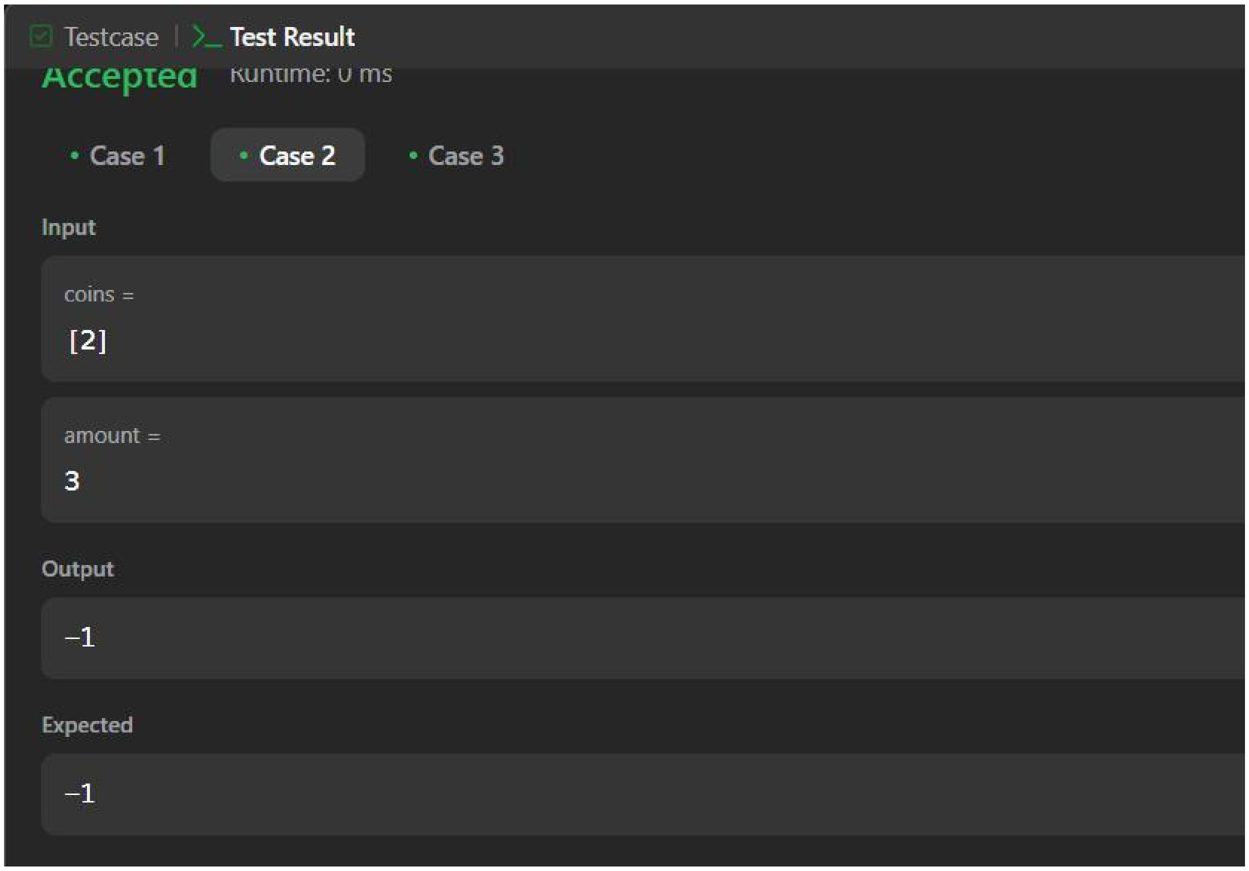
- **Coin Change:**

You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money.
Return the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return -1.
You may assume that you have an infinite number of each kind of coin.

- **Code:**

```
class Solution {
    public int coinChange(int[] coins, int amount) {
        int[] dp = new int[amount + 1];
        Arrays.fill(dp, amount + 1);
        dp[0] = 0;
        for (int coin : coins) {
            for (int i = coin; i <= amount; i++) {
                dp[i] = Math.min(dp[i], dp[i - coin] + 1);
            }
        }

        return dp[amount] == amount + 1 ? -1 : dp[amount];
    }
}
```

➢ **Output:**



```
Input
coins =
[2]
amount =
3
Output
-1
Expected
-1
```

4. <u>Problem - 4:</u>

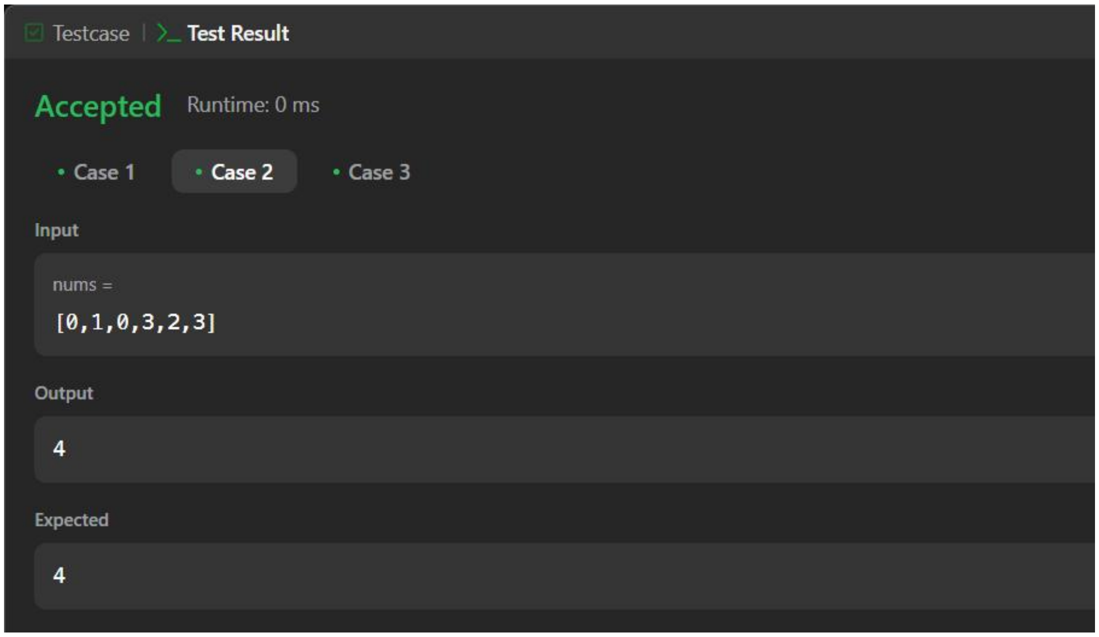
➢ **Longest Increasing Subsequence:**

Given an integer array nums, return the length of the longest strictly increasing subsequence..

➢ **Code:**

```
import java.util.Arrays;

class Solution {
    public int lengthOfLIS(int[] nums) {
        int n = nums.length;
        int[] dp = new int[n];
        Arrays.fill(dp, 1);
        int maxLIS = 1;

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

- **Output:**



Testcase | Test Result

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

```
nums =
[0,1,0,3,2,3]
```

Output

```
4
```

Expected

```
4
```

5. <u>**Problem - 5:**</u>

- **Maximum Product Subarray:**

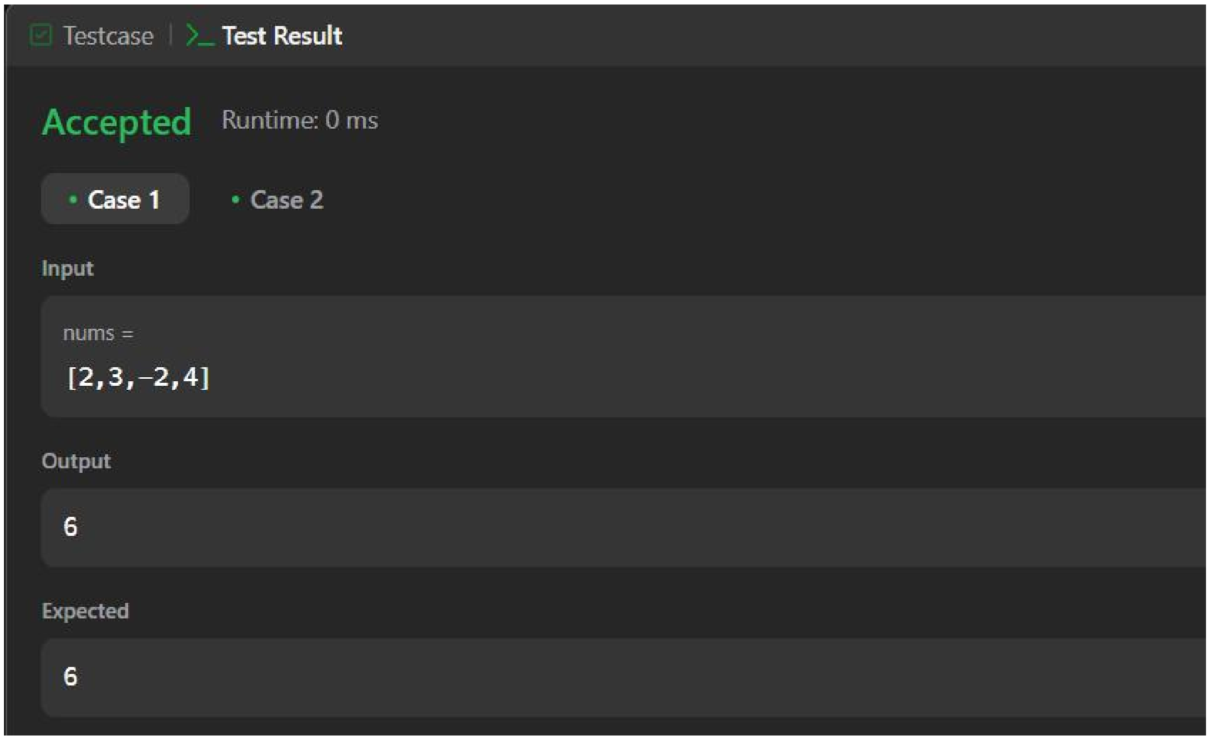
Given an integer array nums, find a subarray that has the largest product, and return the product.
The test cases are generated so that the answer will fit in a 32-bit integer.

- **Code:**

```
class Solution {
    public int maxProduct(int[] nums) {
        int n = nums.length;
        int maxProd = nums[0], minProd = nums[0], result = nums[0];

        for (int i = 1; i < n; i++) {
            if (nums[i] < 0) {
                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;
        }
    }
```

➢ **Output:**



```
Accepted   Runtime: 0 ms
Case 1   Case 2
Input
nums =
[2,3,-2,4]
Output
6
Expected
6
```

6. **<u>Problem - 6:</u>**

➢ **Decode Ways:**

You have intercepted a secret message encoded as a string of numbers. The message is decoded via the following mapping:
"1" -> 'A'
"2" -> 'B'

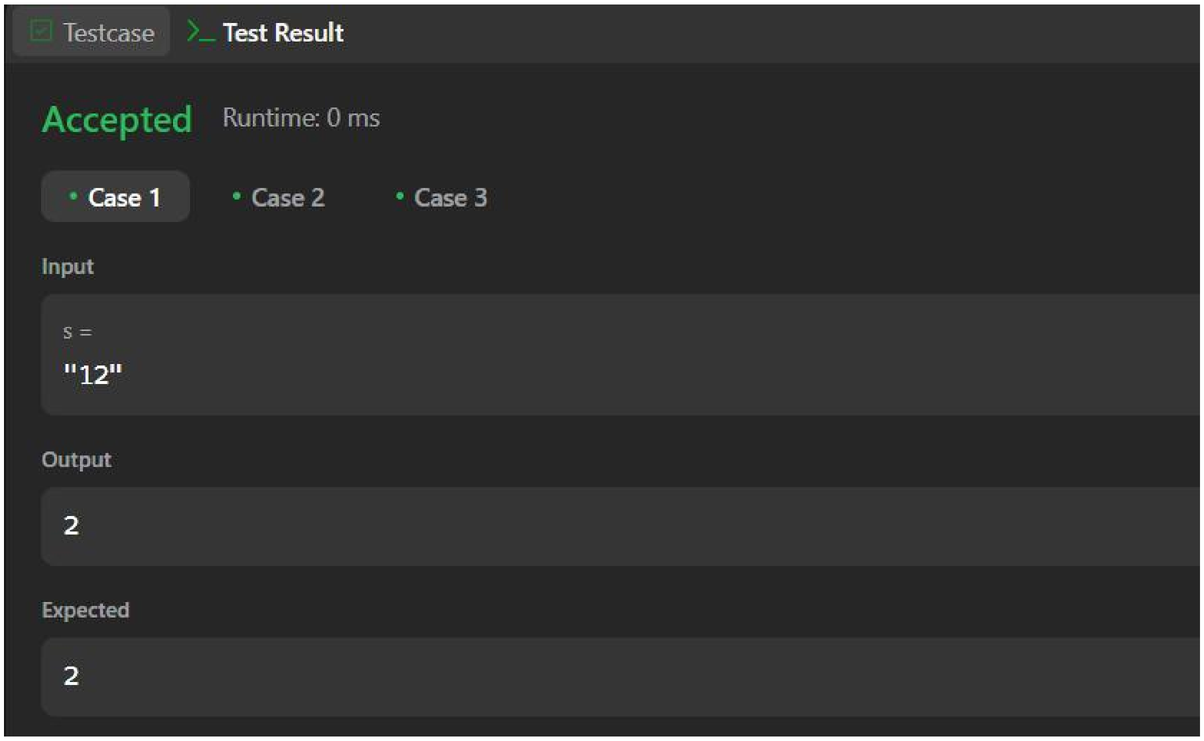
"25" -> 'Y'
"26" -> 'Z'
However, while decoding the message, you realize that there are many different ways you can decode the message because some codes are contained in other codes ("2" and "5" vs "25").
For example, "11106" can be decoded into:
"AAJF" with the grouping (1, 1, 10, 6)
"KJF" with the grouping (11, 10, 6)
The grouping (1, 11, 06) is invalid because "06" is not a valid code (only "6" is valid).
Note: there may be strings that are impossible to decode.

Given a string s containing only digits, return the number of ways to decode it. If the entire string cannot be decoded in any valid way, return 0.
The test cases are generated so that the answer fits in a 32-bit integer.

➢ **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[] dp = new int[n + 1];
        dp[0] = 1;
        dp[1] = s.charAt(0) != '0' ? 1 : 0;

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

            if (oneDigit >= 1) dp[i] += dp[i - 1];
            if (twoDigits >= 10 && twoDigits <= 26) dp[i] += dp[i - 2];
        }

        return dp[n];
    }
}
```

➢ **Output:**



Testcase | Test Result

Accepted Runtime: 0 ms

Case 1 · Case 2 · Case 3

Input

```
s =
"12"
```

Output

```
2
```

Expected

```
2
```

7. **Problem - 7:**

➢ **Perfect Squares:**

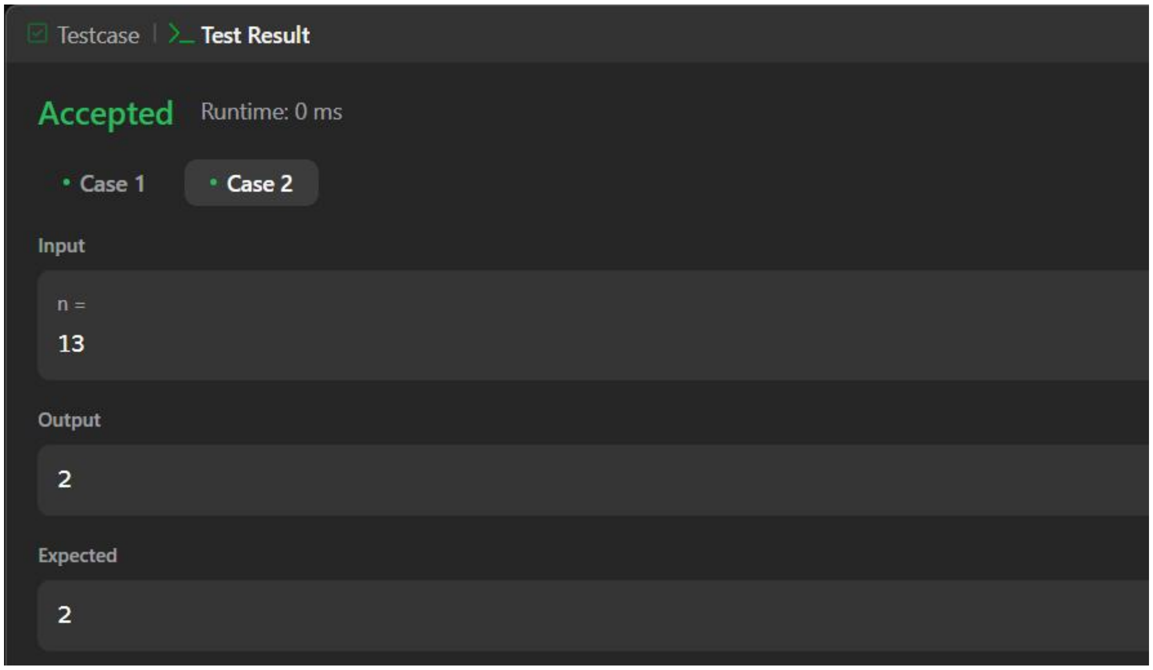
Given an integer n, return the least number of perfect square numbers that sum to n. A perfect square is an integer that is the square of an integer; in other words, it is the product of some integer with itself. For example, 1, 4, 9, and 16 are perfect squares while 3 and 11 are not.
.

➢ **Code:**

```
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;
```

```
            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];
        }
    }
```

➢ **OutPut:**



```
Accepted   Runtime: 0 ms
Input
n =
13
Output
2
Expected
2
```

8. **Problem - 8:**

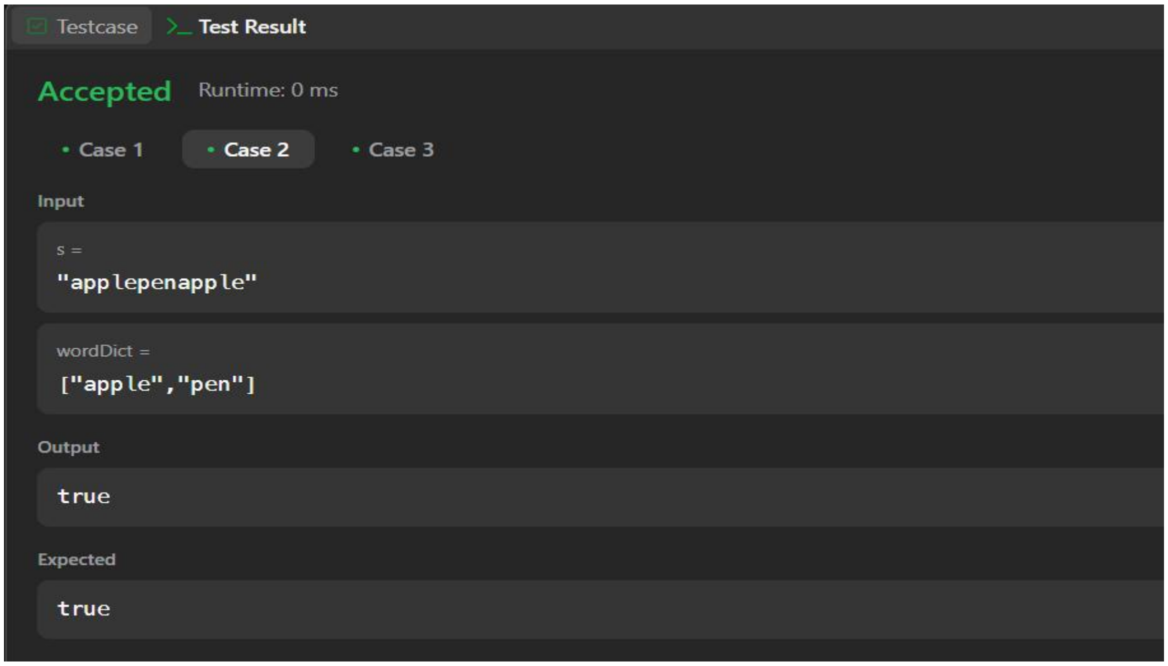
➢ **Word Break:**

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. Note that the same word in the dictionary may be reused multiple times in the segmentation.

➤ **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;
        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];
    }
}
```

➤ **OutPut:**



```
Accepted    Runtime: 0 ms

• Case 1   • Case 2   • Case 3

Input
s =
"applepenapple"
wordDict =
["apple","pen"]

Output
true

Expected
true
```

❖ **Leraning Outcomes:**

- ➢ Understand Dynamic Programming (DP) techniques for solving optimization problems efficiently.
- ➢ Learn how to handle subproblems and overlapping substructure using bottom-up DP.
- ➢ Implement state transitions and recurrence relations in problems like Coin Change, LIS, and Word Break.
- ➢ Apply space and time complexity analysis to optimize solutions, including $O(n^2)$ DP vs. $O(n \log n)$ approaches.
- ➢ Utilize HashSets and Binary Search to further enhance efficiency in word segmentation and sequence problems.