Experiment 7

Student Name: KHUSHBOO **UID: 22BET10197**
Branch: BE-IT **Section/Group: 22BET_IOT_701/A**
Semester: 6th **Date of Performance: 19-03-25**
Subject Name: AP-II **Subject Code: 22ITP-351**

1. **Aim:** Determine if a given string can be segmented into a space-separated sequence of words from a dictionary.

2. Objectives:

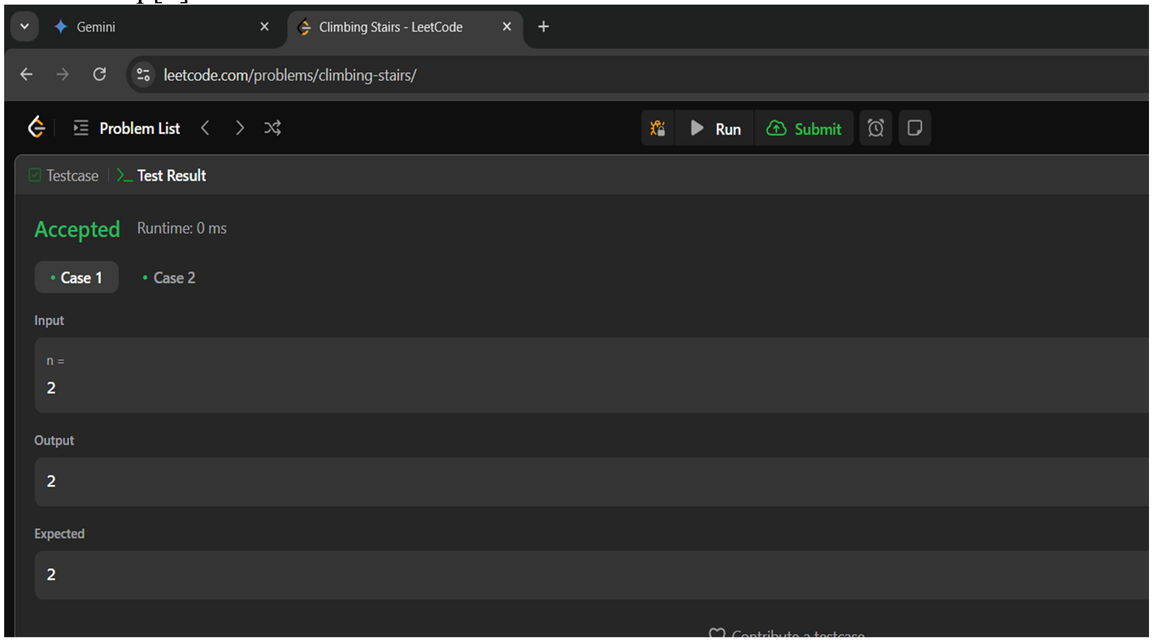
- Create a set of dictionary words for efficient lookup.
- Use dynamic programming to track segmentability of substrings.
- Iterate through the string to check for valid word breaks.
- Return true if the entire string can be segmented, false otherwise.

3. Code:

1. Climbing Stairs (CW)

- **Problem Description:**
  - 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?
- **Algorithm:**
  - This problem can be solved using dynamic programming.
  - Let dp[i] be the number of ways to climb to the i-th step.
  - dp[i] = dp[i-1] + dp[i-2] (because you can reach the i-th step from the (i-1)-th step or the (i-2)-th step).
  - Base cases: dp[1] = 1, dp[2] = 2.
- **Code (Python)**

```
def climbStairs(n: int) -> int:
    if n <= 2:
        return n
    dp = [0] * (n + 1)
    dp[1] = 1
    dp[2] = 2
    for i in range(3, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2]
    return dp[n]
```


Gemini
Climbing Stairs - LeetCode
leetcode.com/problems/climbing-stairs/
Problem List
Run
Submit
Testcase
Test Result

Accepted Runtime: 0 ms

Case 1 Case 2

Input

n =
2

Output

2

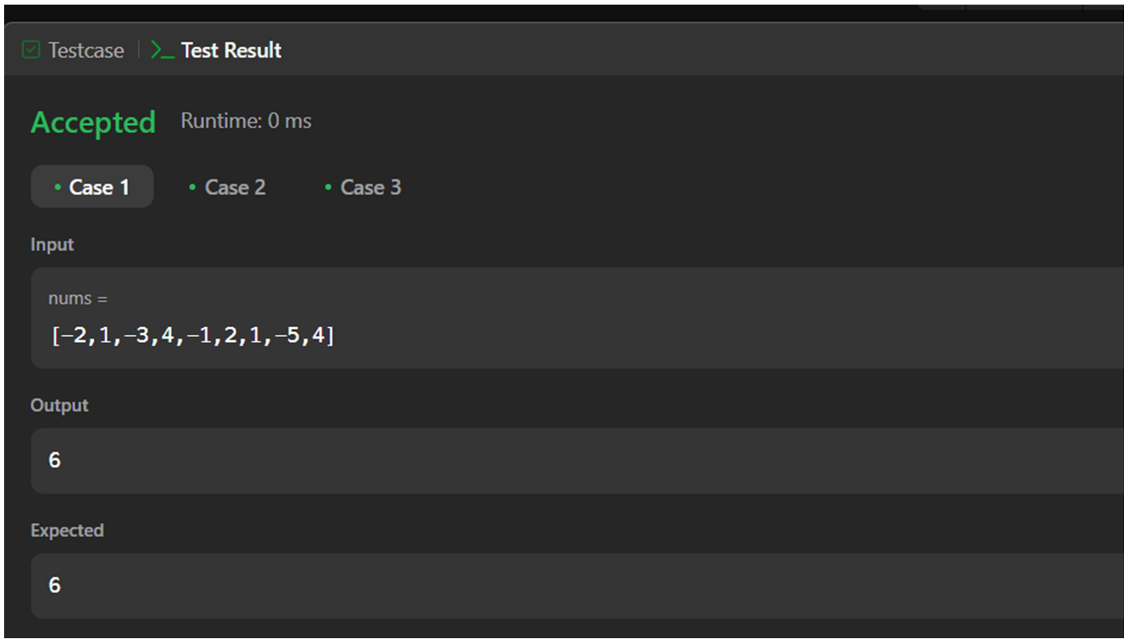
Expected

2

2. Maximum Subarray (CW)

- **Problem Description:**
    - Given an integer array nums, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum.
- **Algorithm:**
    - Use Kadane's algorithm.
    - Use a variable max_so_far to track the maximum sum encountered so far.
    - Use a variable current_max to track the maximum sum ending at the current position.
    - Iterate through the nums array.
    - Update current_max with the maximum of nums[i] and current_max + nums[i].
    - Update max_so_far with the maximum of max_so_far and current_max.
- **Code (Python):**

```
def maxSubArray(nums: list[int]) -> int:
    max_so_far = nums[0]
    current_max = nums[0]
    for i in range(1, len(nums)):
        current_max = max(nums[i], current_max + nums[i])
        max_so_far = max(max_so_far, current_max)
    return max_so_far
```



Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

```
nums =
[-2,1,-3,4,-1,2,1,-5,4]
```

Output

```
6
```

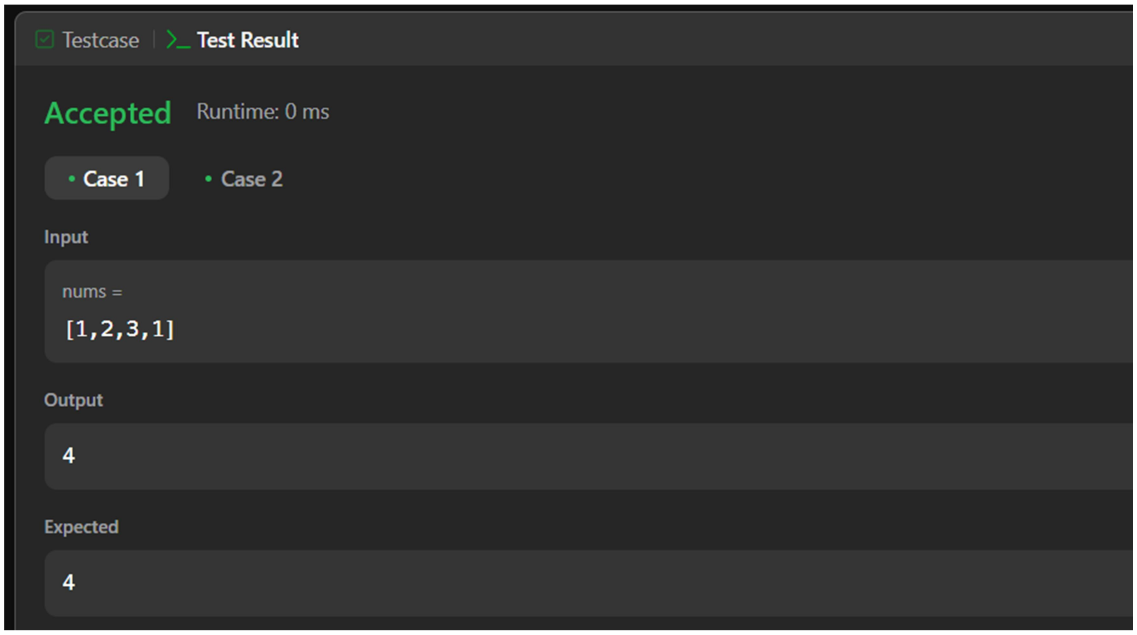
Expected

```
6
```

3. **House Robber (HW)**

- **Problem Description:**
  - You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and it will automatically contact the police if two adjacent houses were broken into on the same night.
  - Given an integer array nums representing the amount of money of each house, return the maximum amount of money you can rob tonight without alerting the police.
- **Algorithm:**
  - Use dynamic programming.
  - Let dp[i] be the maximum amount of money you can rob up to the i-th house.
  - dp[i] = max(dp[i-2] + nums[i], dp[i-1])
  - Base cases: dp[0] = nums[0], dp[1] = max(nums[0], nums[1]).
- **Code (Python):**

```
def rob(nums: list[int]) -> int:
    if not nums:
        return 0
    if len(nums) == 1:
        return nums[0]
    dp = [0] * len(nums)
    dp[0] = nums[0]
    dp[1] = max(nums[0], nums[1])
    for i in range(2, len(nums)):
        dp[i] = max(dp[i - 2] + nums[i], dp[i - 1])
    return dp[len(nums) - 1]
```

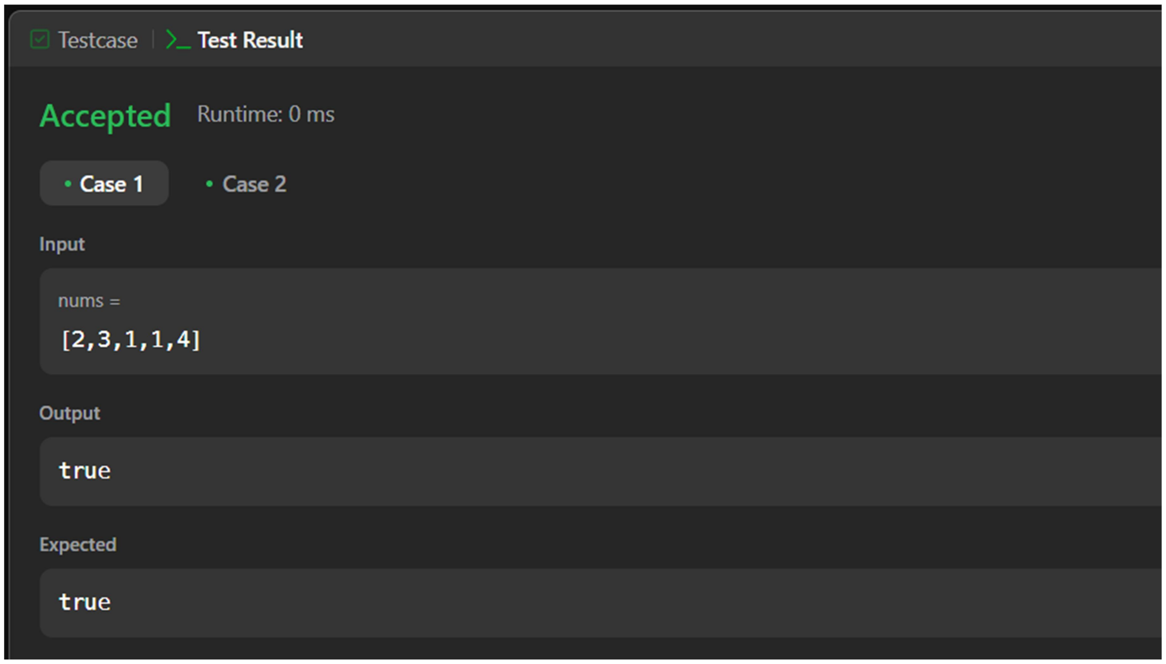


```
Accepted   Runtime: 0 ms
Case 1   Case 2
Input
nums =
[1,2,3,1]
Output
4
Expected
4
```

4. Jump Game (HW)

- **Problem Description:**
    - You are given an integer array nums. You are initially positioned at the array's first index.
    - Each element in the array represents your maximum jump length at that position.
    - Return true if you can reach the last index, or false otherwise.
- **Algorithm:**
    - Use a greedy approach.
    - Keep track of the farthest reachable index.
    - Iterate through the array.
    - If the current index is beyond the farthest reachable index, return false.
    - Update the farthest reachable index.
    - If the farthest reachable index is greater than or equal to the last index, return true.
- **Code (Python):**

```python
def canJump(nums: list[int]) -> bool:
    farthest = 0
    for i in range(len(nums)):
        if i > farthest:
            return False
        farthest = max(farthest, i + nums[i])
        if farthest >= len(nums) - 1:
            return True
    return True
```



Accepted Runtime: 0 ms

Case 1 Case 2

Input

```
nums =
[2,3,1,1,4]
```

Output

```
true
```

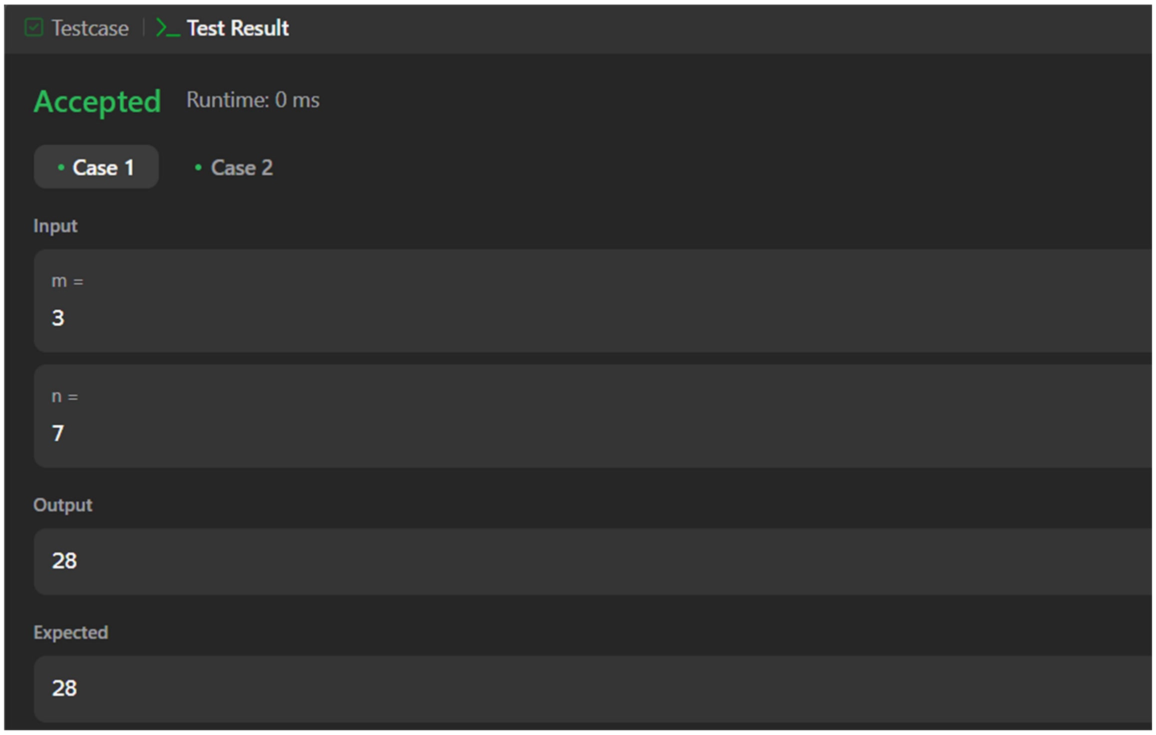
Expected

```
true
```

5. Unique Paths (CW)

- **Problem Description:**
  - 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.
- **Algorithm:**
  - Use dynamic programming.
  - Let dp[i][j] be the number of unique paths to reach cell (i, j).
  - dp[i][j] = dp[i-1][j] + dp[i][j-1]
  - Base cases: dp[0][j] = 1, dp[i][0] = 1.
- **Code (Python)**

```
def uniquePaths(m: int, n: int) -> int:
    dp = [[0] * n for _ in range(m)]
    for i in range(m):
        dp[i][0] = 1
    for j in range(n):
        dp[0][j] = 1
    for i in range(1, m):
        for j in range(1, n):
            dp[i][j] = dp[i - 1][j] + dp[i][j - 1]
    return dp[m - 1][n - 1]
```


Testcase | Test Result

Accepted Runtime: 0 ms

• Case 1 • Case 2

Input

```
m =
3
```

```
n =
7
```

Output

```
28
```

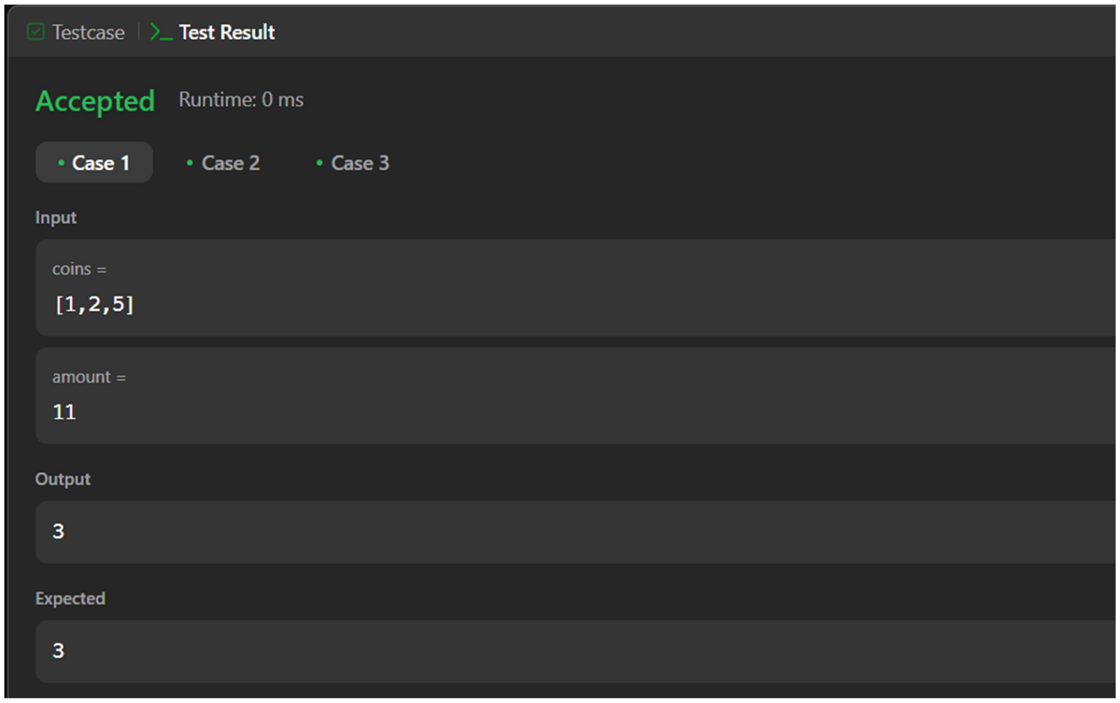
Expected

```
28
```

6. Coin Change (CW)

- **Problem Description:**
  - 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.
- **Algorithm:**
  - Use dynamic programming.
  - Let dp[i] be the minimum number of coins needed to make up amount i.
  - dp[i] = min(dp[i-coin] + 1) for all coin in coins if i-coin >=0
  - Base case: dp[0] = 0.
  - Initialize dp with infinity.
- **Code (Python):**

```python
def coinChange(coins: list[int], amount: int) -> int:
    dp = [float('inf')] * (amount + 1)
    dp[0] = 0
    for i in range(1, amount + 1):
        for coin in coins:
            if i - coin >= 0:
                dp[i] = min(dp[i], dp[i - coin] + 1)
    return dp[amount] if dp[amount] != float('inf') else -1
```

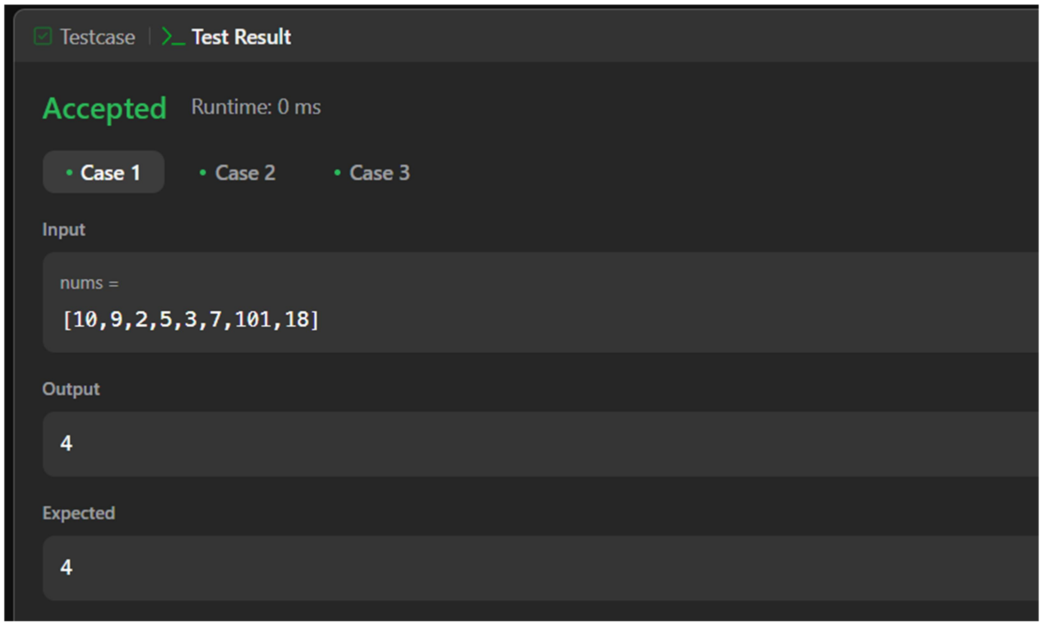


```
Accepted    Runtime: 0 ms
• Case 1   • Case 2   • Case 3
Input
coins =
[1,2,5]
amount =
11
Output
3
Expected
3
```

7. Longest Increasing Subsequence (CW)

- **Problem Description:**
    - Given an integer array nums, return the length of the longest strictly increasing subsequence.
- **Algorithm:**
    - Use dynamic programming.
    - Let dp[i] be the length of the longest increasing subsequence ending at nums[i].
    - dp[i] = max(dp[j] + 1) for all j < i if nums[i] > nums[j]
    - Initialize dp array with 1.
- **Code (Python):**

```
def lengthOfLIS(nums: list[int]) -> int:
    if not nums:
        return 0
    dp = [1] * len(nums)
    for i in range(1, len(nums)):
        for j in range(i):
            if nums[i] > nums[j]:
                dp[i] = max(dp[i], dp[j] + 1)
    return max(dp)
```

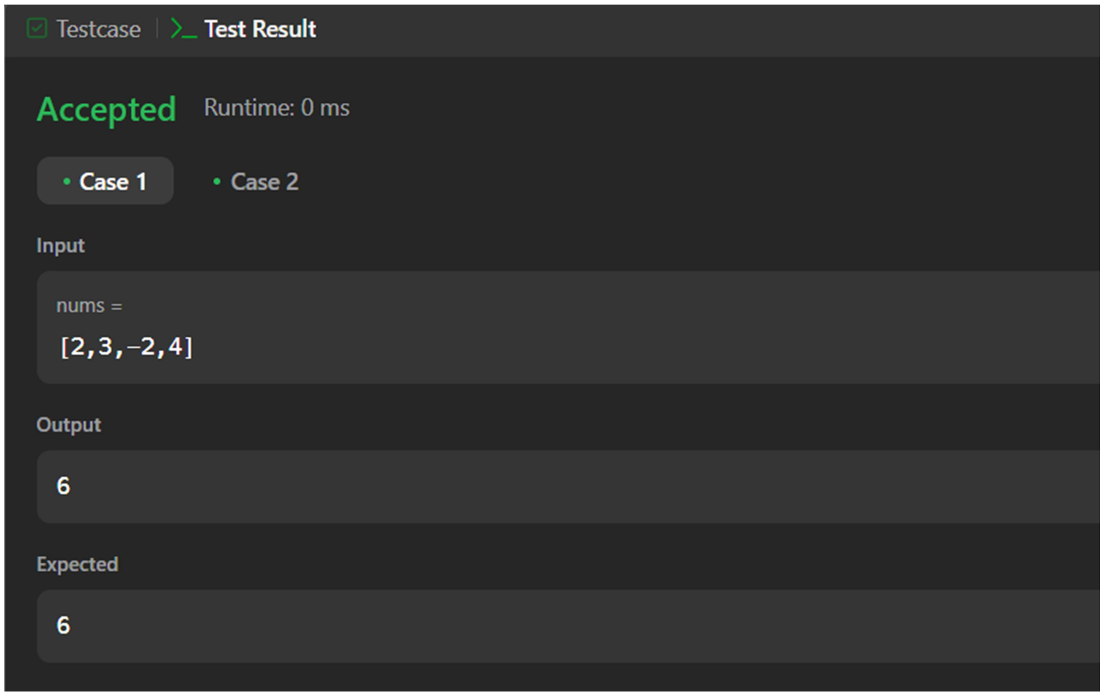


```
Accepted   Runtime: 0 ms

Input
nums =
[10,9,2,5,3,7,101,18]

Output
4

Expected
4
```

8. Maximum Product Subarray (HW)

- **Problem Description:**
  - Given an integer array nums, find the contiguous subarray within an array (containing at least one number) which has the largest product.
- **Algorithm:**
  - Use dynamic programming.
  - Keep track of the maximum and minimum product ending at each position.
  - max_dp[i] = max(nums[i], nums[i] * max_dp[i-1], nums[i] * min_dp[i-1])
  - min_dp[i] = min(nums[i], nums[i] * max_dp[i-1], nums[i] * min_dp[i-1])
- **Code (Python)**

```python
def maxProduct(nums: list[int]) -> int:
    if not nums:
        return 0
    max_dp = [0] * len(nums)
    min_dp = [0] * len(nums)
    max_dp[0] = nums[0]
    min_dp[0] = nums[0]
    result = nums[0]
    for i in range(1, len(nums)):
        max_dp[i] = max(nums[i], nums[i] * max_dp[i - 1], nums[i] * min_dp[i - 1])
        min_dp[i] = min(nums[i], nums[i] * max_dp[i - 1], nums[i] * min_dp[i - 1])
        result = max(result, max_dp[i])
    return result
```



Accepted Runtime: 0 ms

Case 1 Case 2

Input

```
nums =
[2,3,-2,4]
```

Output

```
6
```

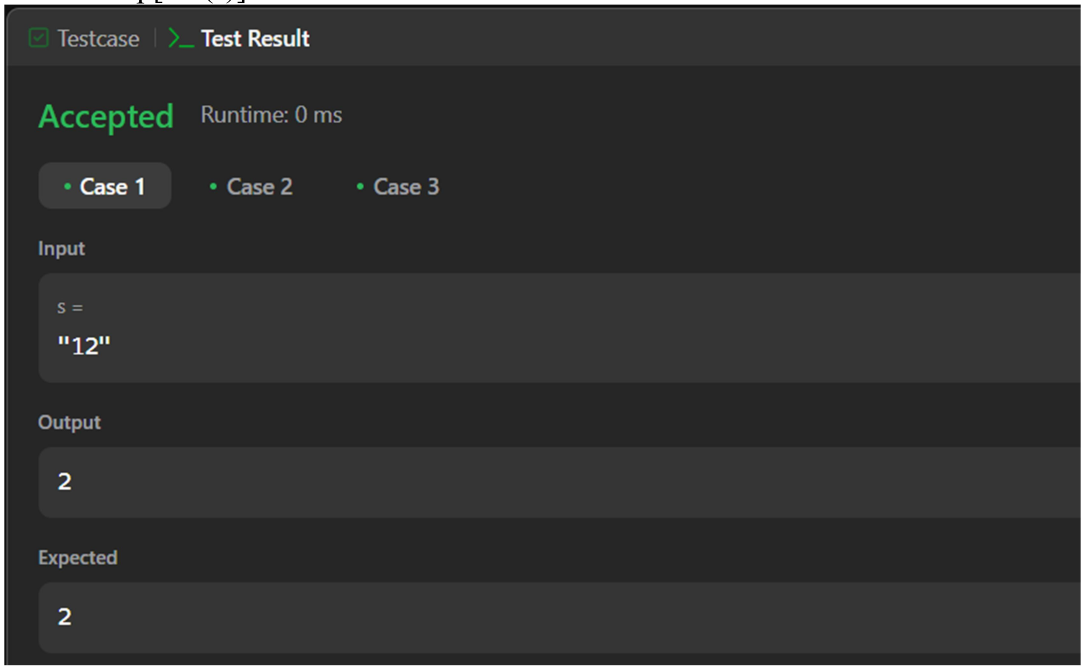
Expected

```
6
```

9. Decode Ways (CW)

- **Problem Description:**
  - A message containing letters from A-Z can be encoded into numbers using the following mapping:
    - 'A' -> "1"
    - 'B' -> "2"
    - ...
    - 'Z' -> "26"
  - Given a string s containing only digits, return the number of ways to decode it.
- **Algorithm:**
  - Use dynamic programming.
  - Let dp[i] be the number of ways to decode s[:i].
  - If s[i-1] is not '0', dp[i] += dp[i-1].
  - If s[i-2:i] is between "10" and "26", dp[i] += dp[i-2].
  - base cases: dp[0] = 1, dp[1] = 1 if s[0] != '0' else 0
- **Code (Python):**

```python
def numDecodings(s: str) -> int:
    if not s or s[0] == '0':
        return 0
    dp = [0] * (len(s) + 1)
    dp[0] = 1
    dp[1] = 1
    for i in range(2, len(s) + 1):
        if s[i - 1] != '0':
            dp[i] += dp[i - 1]
        if '10' <= s[i - 2:i] <= '26':
            dp[i] += dp[i - 2]
    return dp[len(s)]
```



Testcase | Test Result

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

s =

```
"12"
```

Output

```
2
```

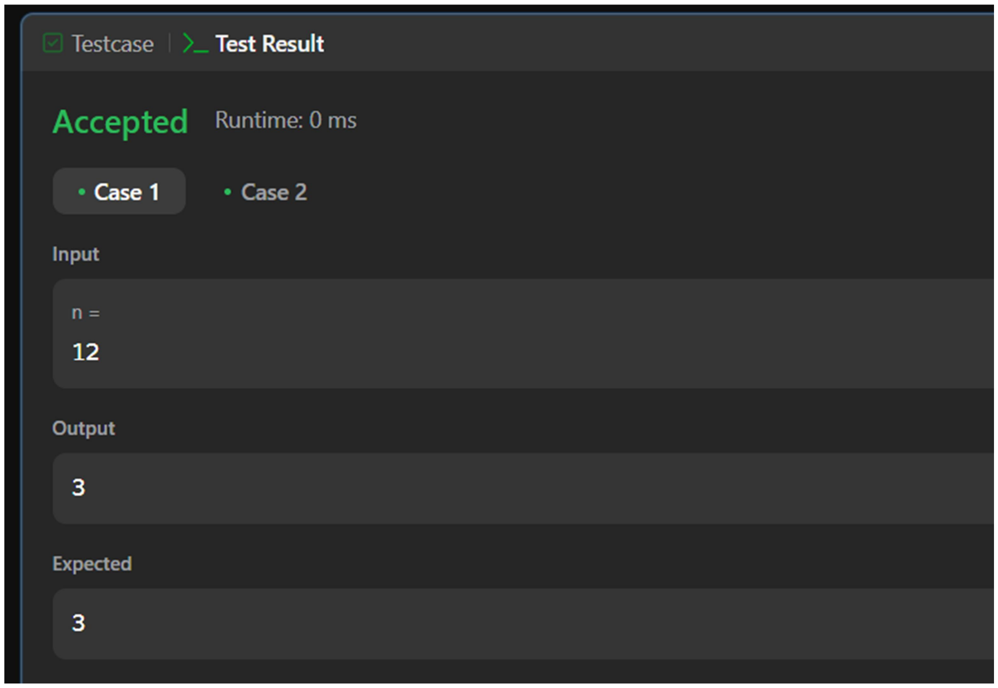
Expected

```
2
```

10. Perfect Squares (HW)

- **Problem Description:**
  - Given an integer n, return the least number of perfect square numbers that sum to n.
- **Algorithm:**
  - Use dynamic programming.
  - dp[i] represents the least number of perfect squares that sum to i.
  - dp[i] = min(dp[i-j*j] + 1) for all j if j*j <= i
  - Base case: dp[0] = 0.
- **Code (Python):**

```
def numSquares(n: int) -> int:
    dp = [float('inf')] * (n + 1)
    dp[0] = 0
    for i in range(1, n + 1):
        j = 1
        while j * j <= i:
            dp[i] = min(dp[i], dp[i - j * j] + 1)
            j += 1
    return dp[n]
```



Accepted Runtime: 0 ms

Case 1 Case 2

Input

```
n =
12
```

Output

```
3
```

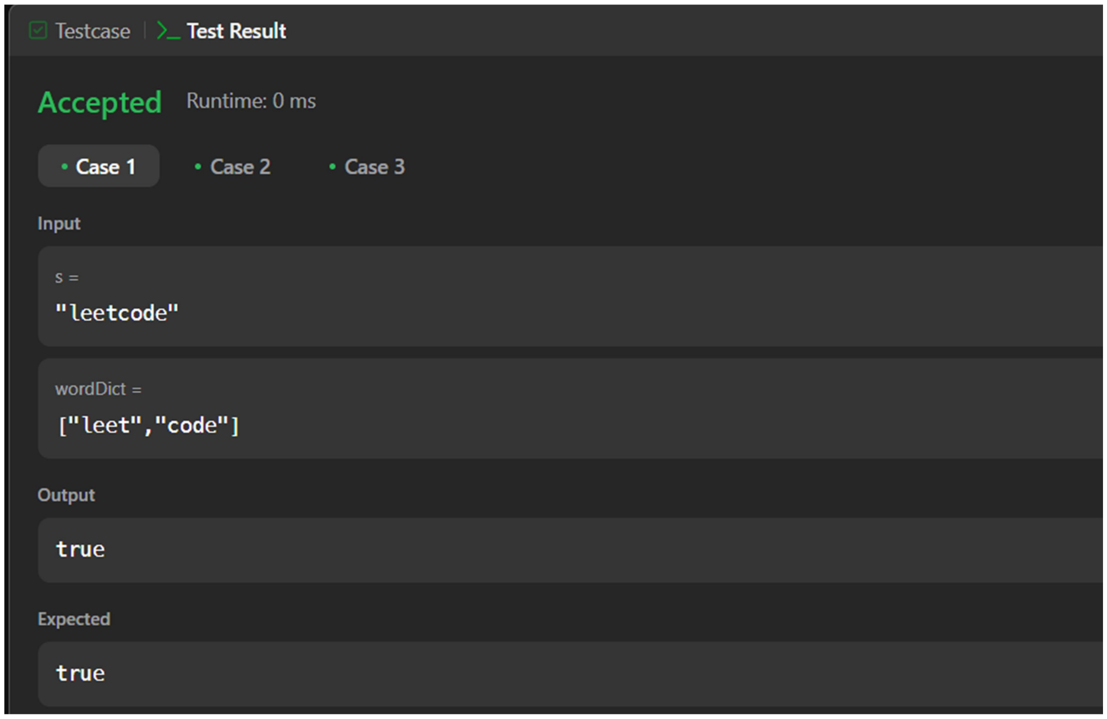
Expected

```
3
```

11. Word Break (CW)

- **Problem Description:**
  - 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.
- **Algorithm:**
  - Use dynamic programming.
  - dp[i] represents whether s[:i] can be segmented into words from wordDict.
  - dp[i] = True if dp[j] is True and s[j:i] is in wordDict for some j < i.
  - Base case: dp[0] = True.
- **Code (Python):**

```
def wordBreak(s: str, wordDict: list[str]) -> bool:
    dp = [False] * (len(s) + 1)
    dp[0] = True
    for i in range(1, len(s) + 1):
        for j in range(i):
            if dp[j] and s[j:i] in wordDict:
                dp[i] = True
                break
    return dp[len(s)]
```



Testcase | >_ Test Result

Accepted Runtime: 0 ms

• Case 1 • Case 2 • Case 3

Input

```
s =
"leetcode"
```

```
wordDict =
["leet","code"]
```

Output

```
true
```

Expected

```
true
```