Experiment- 7

Student Name: Kamal Mehta
UID: 22BET10097
Branch: B.E - IT
Section/Group: 22BET-701/A
Semester: 6th
Date of Performance: 19-03-25
Subject Name: AP Lab -2
Subject Code: 22ITP-351

Problem 1: Unique Paths

1. **Problem Statement:** To determine the number of unique paths from the top-left corner to the bottom-right corner of a grid.

2. **Objective:**

I. To Implement a solution that counts paths using combinatorial mathematics or dynamic programming.

II. To develop an algorithm that efficiently computes the minimum coins required.

III. To Implement a dynamic programming solution to find the longest increasing subsequence.

IV. To Develop an efficient algorithm that tracks the maximum and minimum products at each step.

V. To implement Implement a dynamic programming approach to count the decoding ways.

VI. To Explore mathematical properties of perfect squares.

VII. To Implement a dynamic programming solution to check for valid segmentations.

3. Code:

```
class Solution:
    def uniquePaths(self, m: int, n: int) -> int:

        aboveRow = [1] * n
        for _ in range(m - 1):
            currentRow = [1] * n
            for i in range(1, n):
                currentRow[i] = currentRow[i-1] + aboveRow[i]
            aboveRow = currentRow

        return aboveRow[-1]
```

4. Output:

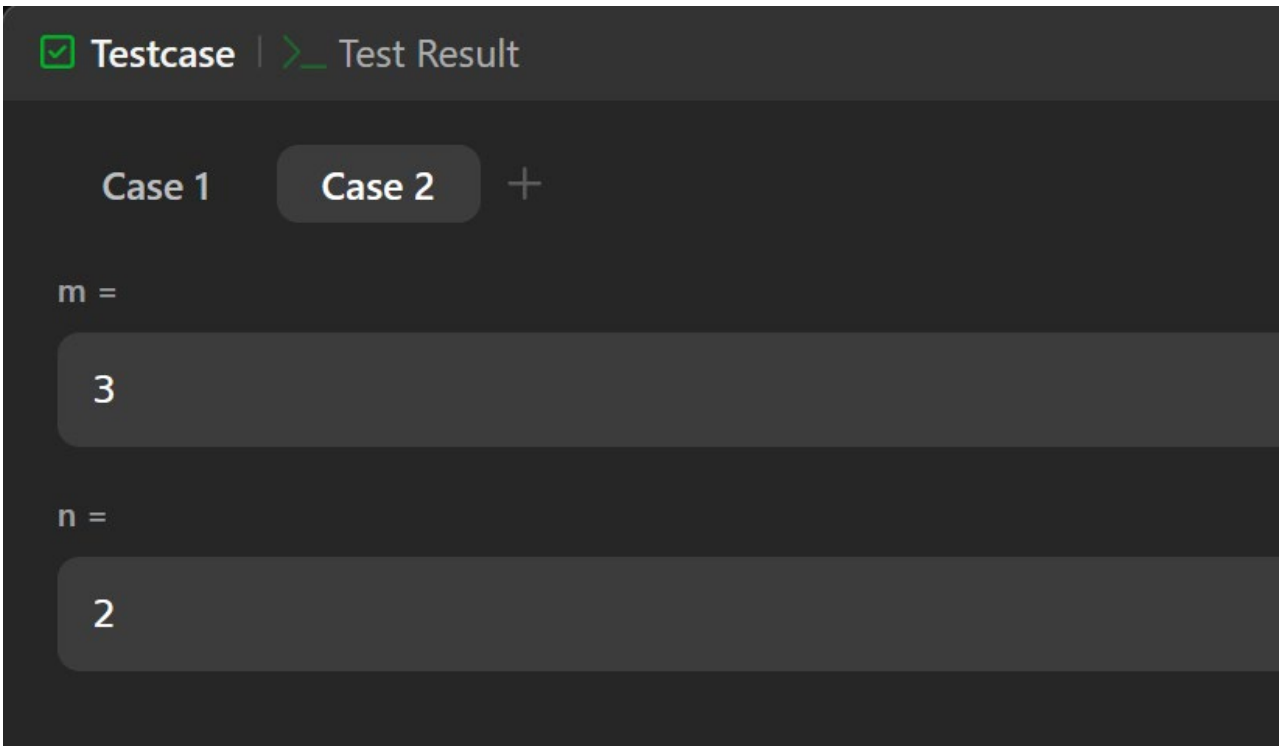

Fig 1: Output for Problem 1

Problem 2: Coin Change

1. **Problem Statement:** To find the minimum number of coins needed to make a certain amount of money using given denominations.

2. **Code:**

```
class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        from functools import lru_cache

        @lru_cache(None)
        def solve(rem):
            if rem == 0:
                return 0
            if rem < 0:
                return float('inf')

            return min(1 + solve(rem - coin) for coin in coins)

        res = solve(amount)
        return res if res != float('inf') else -1
```

3. **Output:**

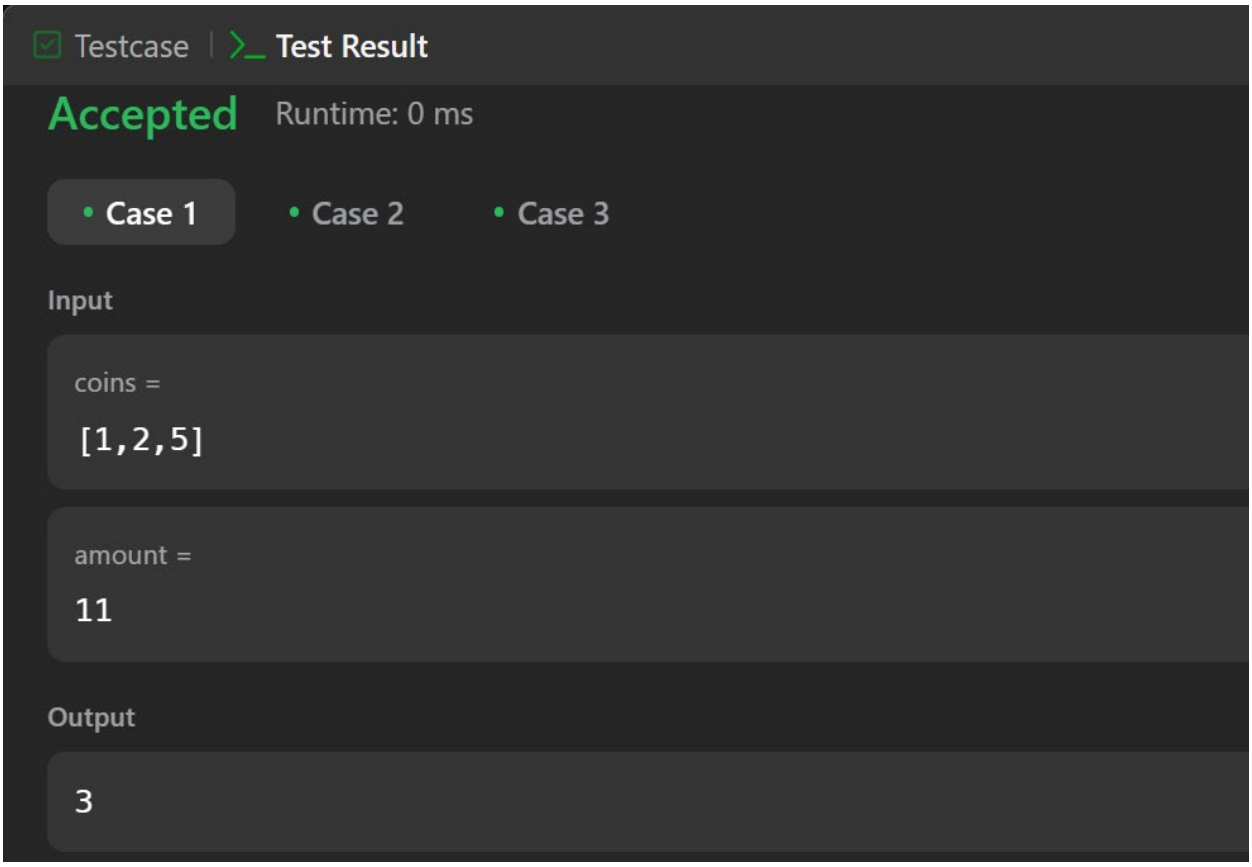

Fig 2: Output for Problem 2

Problem 3: Longest Increasing Subsequence

1. **Problem Statement:** To find the length of the longest subsequence in a sequence of numbers where the subsequence is strictly increasing.

2. **Code:**

```
class Solution:
    def lengthOfLIS(self, nums: List[int]) -> int:
        res = []
        def binary_search(res, n):
            left = 0
            right = len(res) - 1

            while left <= right:
                mid = (left + right) // 2
                if res[mid] == n:
                    return mid
                elif res[mid] > n:
                    right = mid - 1
                else:
                    left = mid + 1
            return left

        for n in nums:
            if not res or res[-1] < n:
```

```
            res.append(n)
        else:
            idx = binary_search(res, n)
            res[idx] = n

    return len(res)
```

3. Output:

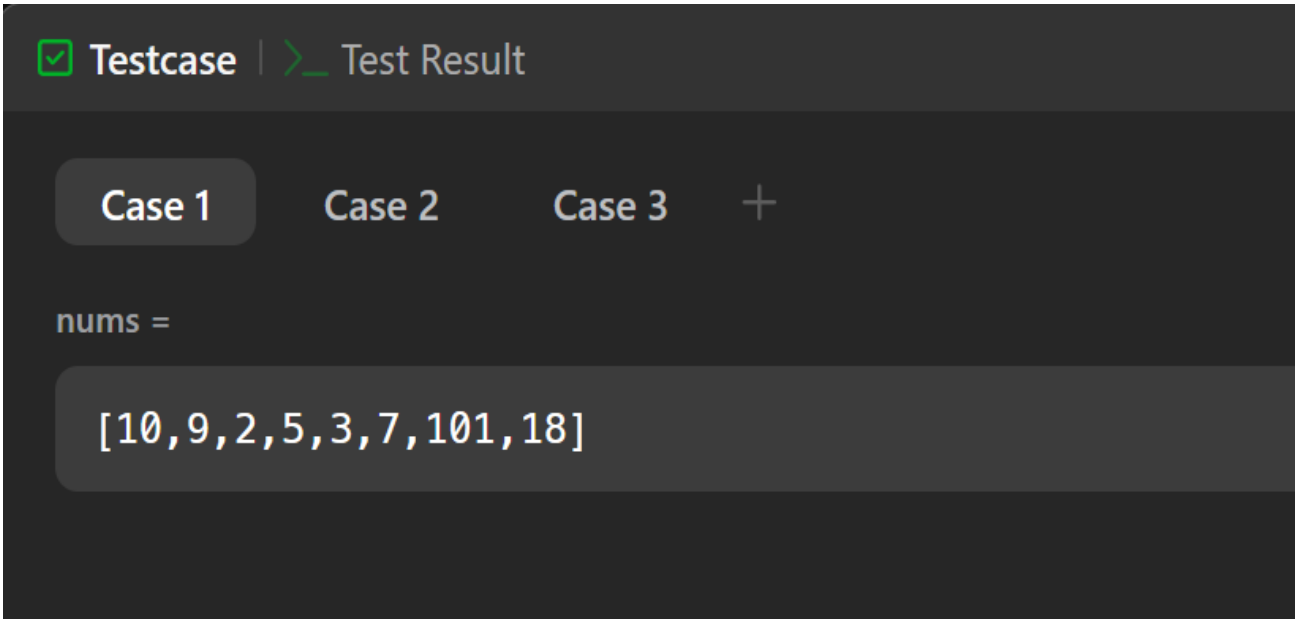

Fig 3: Output for Problem 3

Problem 4: Maximum Product Subarray

1. **Problem Statement:** To find the contiguous subarray within a one-dimensional array of numbers that has the largest product.

2. **Code:**

```
class Solution:

    def maxProduct(self, nums: List[int]) -> int:
        res = max(nums)
        cur_max = cur_min = 1

        for n in nums:
            temp = cur_max * n
            cur_max = max(temp, cur_min * n, n)
            cur_min = min(temp, cur_min * n, n)

            res = max(res, cur_max)

        return res
```

3. Output:

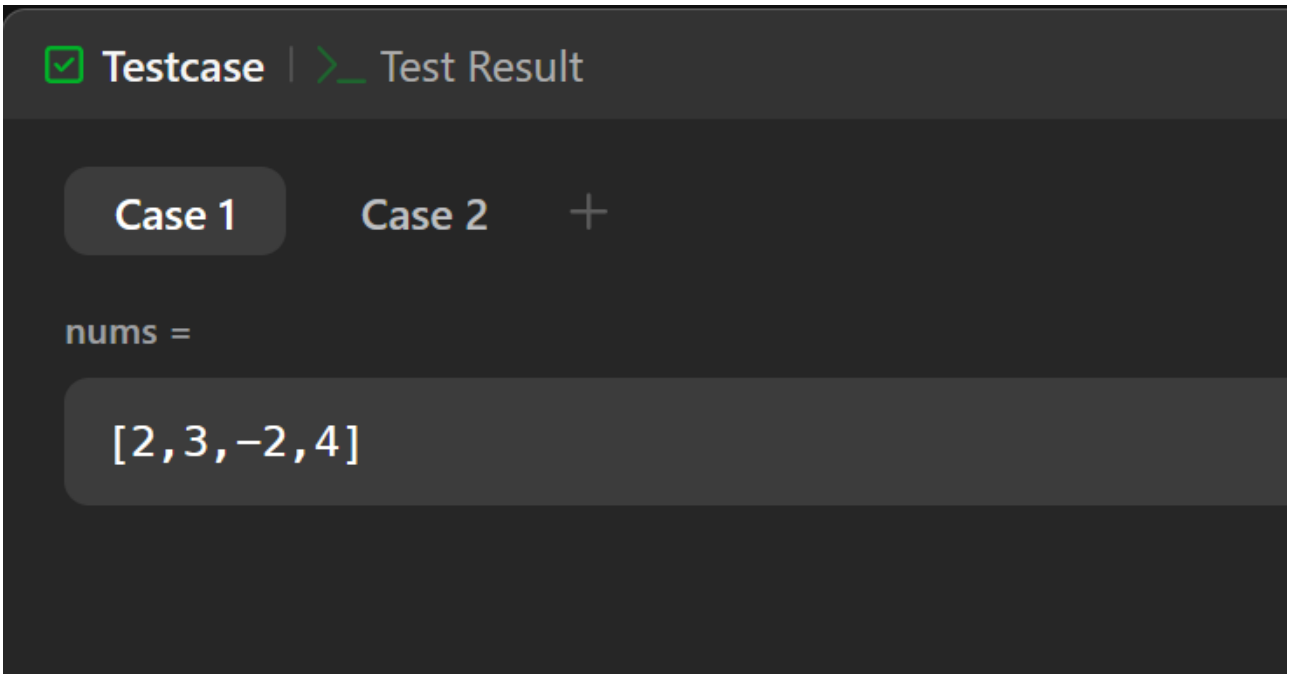

Fig 4: Output for Problem 4

Problem 5: Decode Ways

1. **Problem Statement:** To determine the number of ways to decode a given string of digits into letters.

2. **Code:**

```
class Solution:

    def numDecodings(self, s: str) -> int:
        if s[0] == '0':
            return 0

        n = len(s)
        dp = [0] * (n + 1)
        dp[0], dp[1] = 1, 1

        for i in range(2, n + 1):
            one = int(s[i - 1])
            two = int(s[i - 2:i])

            if 1 <= one <= 9:
                dp[i] += dp[i - 1]
            if 10 <= two <= 26:
                dp[i] += dp[i - 2]

        return dp[n]
```

3. **Output:**

Fig 5: Output for Problem 5

Problem 6: Perfect Squares

1. **Problem Statement:** To find the least number of perfect square numbers that sum up to a given positive integer.

2. **Code:**

```
class Solution:
    def numSquares(self, n: int) -> int:
        dp=[0]*(n+1)
        for i in range(1,n+1):
            dp[i]=i
            for j in range(1,int(math.sqrt(i))+1):
                dp[i]=min(dp[i],dp[i-j*j]+1)
        return dp[n]
```

3. **Output:**

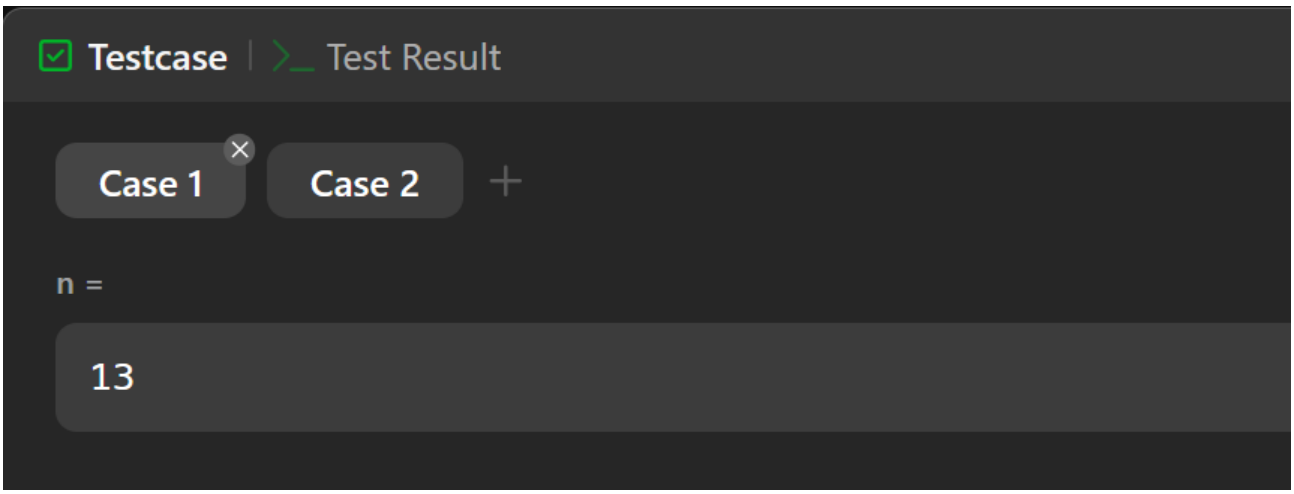

Fig 6: Output for Problem 6

Problem 7: Word Break

1. **Problem Statement:** To determine if a given string can be segmented into a space-separated sequence of one or more dictionary words.

2. **Code:**

```
class Solution:
    def wordBreak(self, s: str, wordDict: List[str]) -> bool:

        def construct(current,wordDict, memo={}):
            if current in memo:
                return memo[current]

            if not current:
                return True

            for word in wordDict:
                if current.startswith(word):
                    new_current = current[len(word):]
                    if construct(new_current,wordDict,memo):
                        memo[current] = True
                        return True

            memo[current] = False
            return False

        return construct(s,wordDict)
```

3. **Output:**

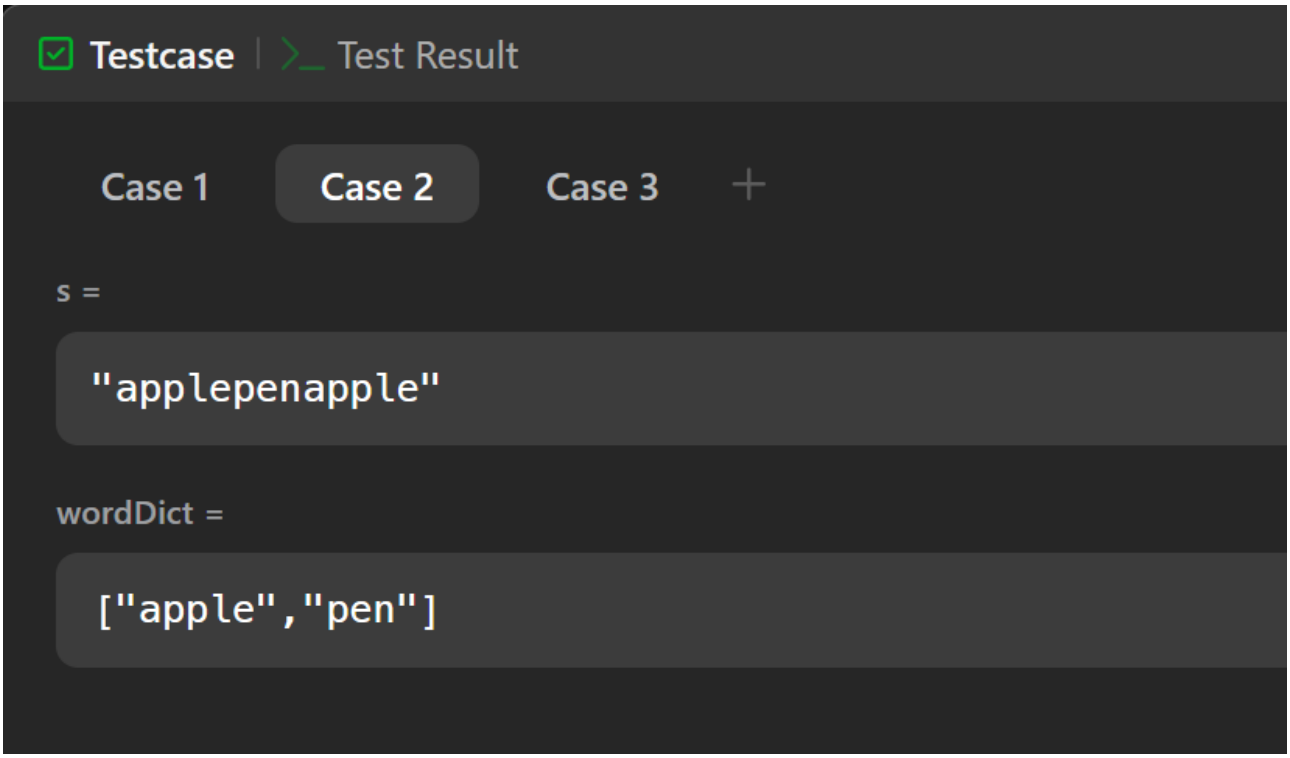

Fig 7: Output for Problem 7

4. Learning Outcome:

1. Developed the ability to analyze complex problems and break them down into manageable components, leading to effective solutions.
2. Gained understanding of dynamic programming techniques, including memoization and tabulation, and learn how to apply them to optimize solutions for various types of problems.
3. Learned to apply combinatorial principles to problems such as counting unique paths and coin change, enhancing mathematical reasoning skills.

4. Acquired skills in designing efficient algorithms, focusing on time and space complexity, and learn how to optimize solutions for better performance.
5. Developed the ability to identify and manage edge cases and constraints in problem statements, ensuring robust and reliable solutions.
6. Explored the use of greedy algorithms and backtracking in problems like word break, enhancing the understanding of different algorithmic strategies.