Experiment-7

Name: Yash Bamankar
UID: 22BET10301
Branch: BE-IT
Section/Group: 22BET_IOT-702/A
Semester: 6th
Date of Performance:20/03/25
Subject Name: AP LAB-II
Subject Code: 22ITP-351

Problem-1

1.Aim:
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?

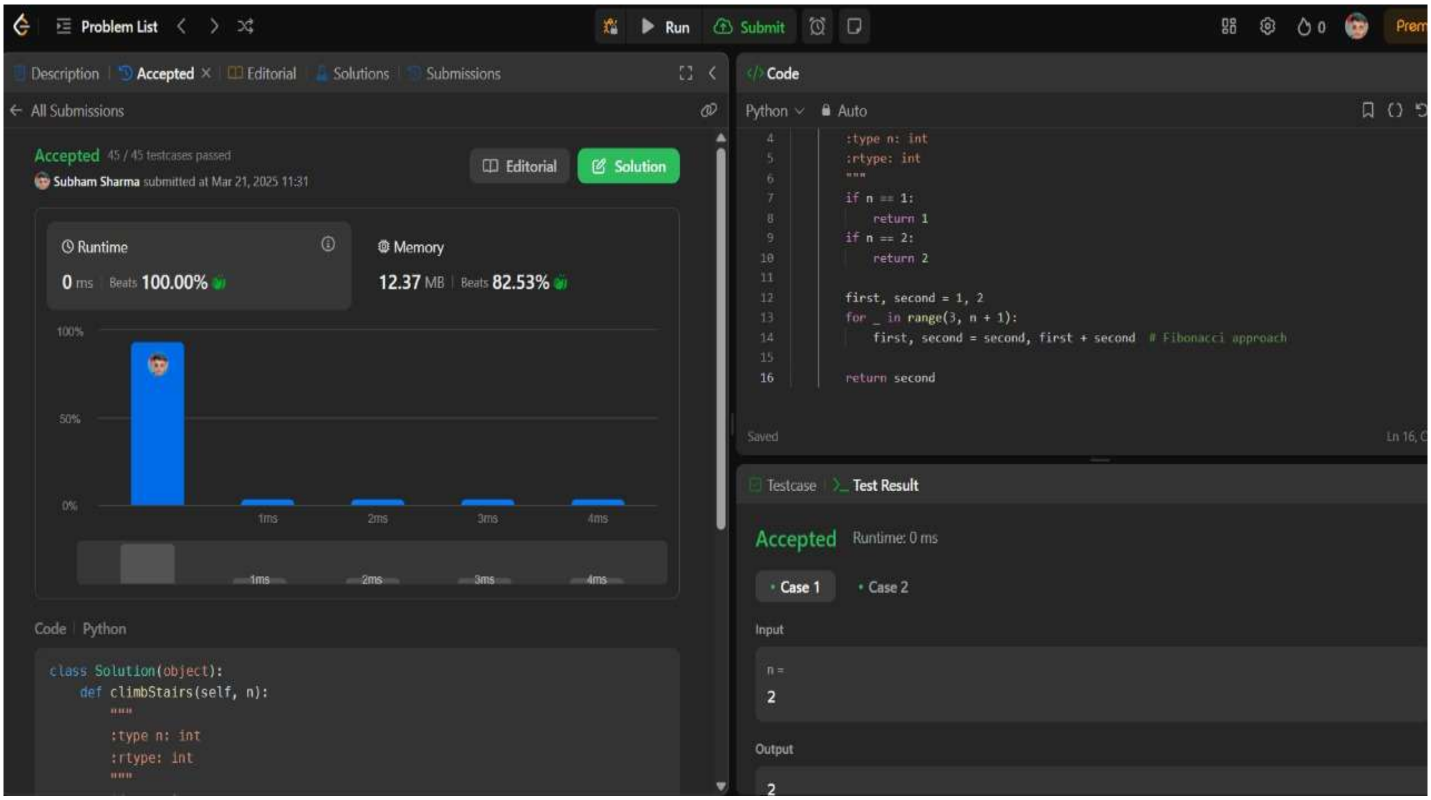
2. Objective:

- Learn how to solve the problem using recursion and dynamic programming.
- Understand the Fibonacci sequence in the context of counting paths.

3. Code:

```
class Solution(object):
    def climbStairs(self, n):

        type n: int
        rtype: int



        if n == 1:

            return 1
        if n == 2:
            return 2

        first, second = 1, 2
        for _ in range(3, n + 1):
            first, second = second, first + second  # Fibonacci approach

        return second
```

4.Output:



5 Learning outcomes:

1.Learn how to break down the problem into smaller subproblems.

2.Implement a recursive solution to find the number of ways to climb stairs.

3.Improve recursive solutions by storing previously computed values.

Fig.1:Climbing Stairs

Problem-2

1. Aim:
Given an integer array nums, find the subarray with the largest sum, and return *its sum*.

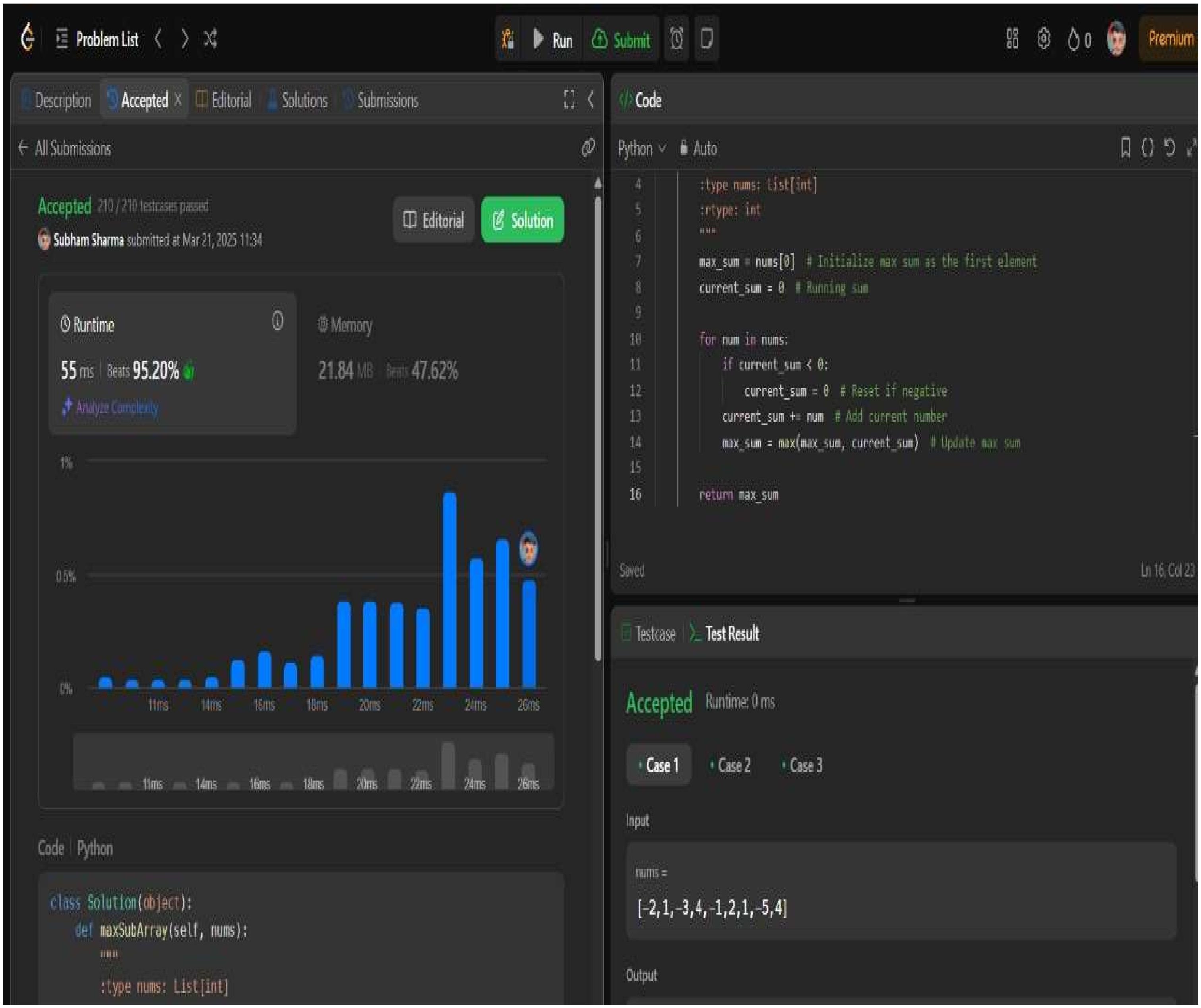
2. Objective:

- Learn how to apply Kadane's algorithm to find the maximum subarray sum.
- Understand the concept of dynamic programming in optimizing subarray problems**.**

3. Code:
```
class Solution(object):
    def maxSubArray(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        max_sum = nums[0]  # Initialize max sum as the first element
        current_sum = 0  # Running sum

        for num in nums:
            if current_sum < 0:
                current_sum = 0  # Reset if negative
            current_sum += num  # Add current number
            max_sum = max(max_sum, current_sum)  # Update max sum



        return max_sum
```

= 0;

Output:



Learning outcomes:

1 . Improve recursive solutions by storing previously computed values.

2.Improve recursive solutions by storing previously computed values.

Fig.2: MaxSubarray

Problem-3

1. Aim:

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.

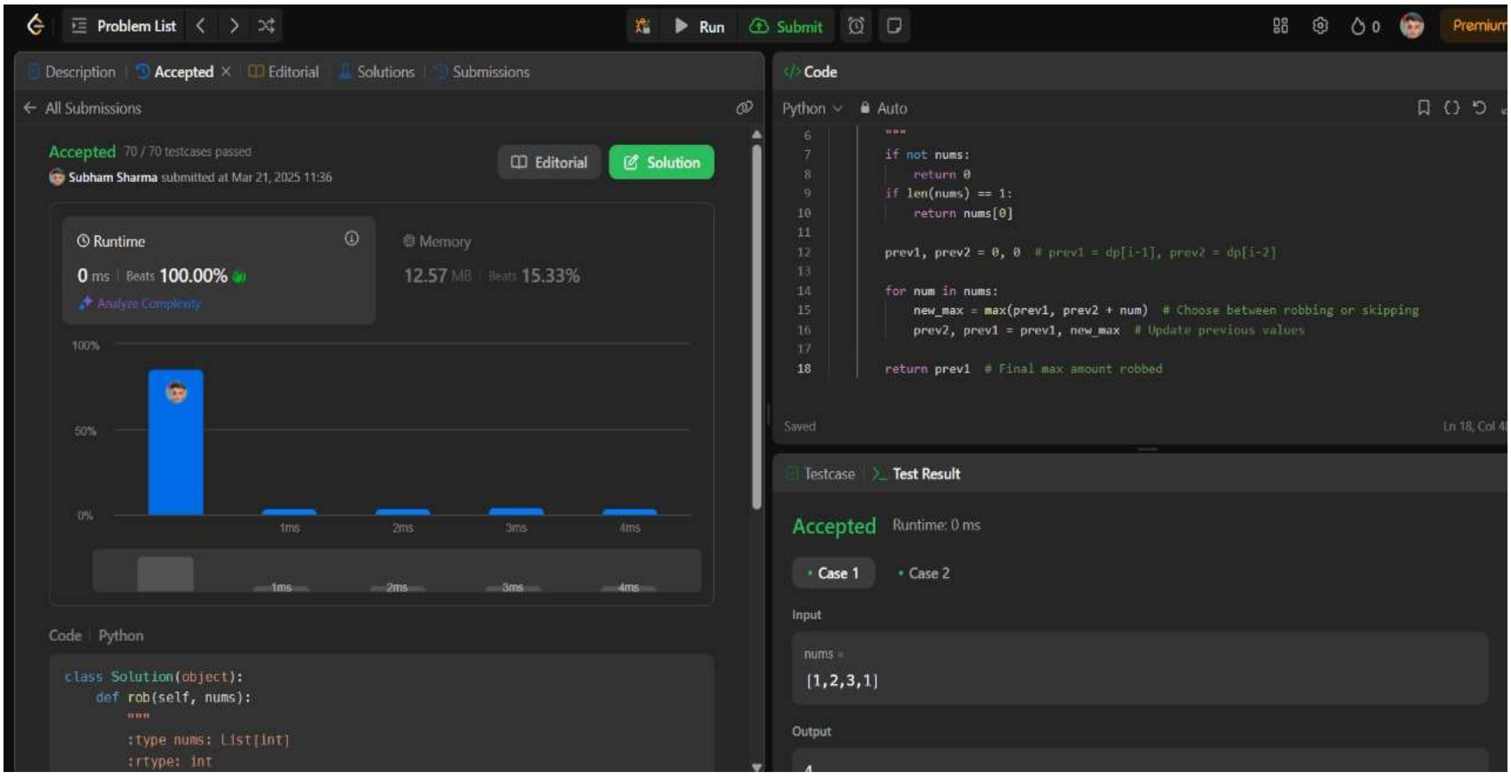
2. Objective:

- Understand dynamic programming to solve problems with adjacent constraints.
- Learn how to optimize decision-making using a rolling sum approach.

3. Code:

```
class Solution(object):
    def rob(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        if not nums:
            return 0
        if len(nums) == 1:
            return nums[0]

        prev1, prev2 = 0, 0  # prev1 = dp[i-1], prev2 = dp[i-2]

        for num in nums:
            new_max = max(prev1, prev2 + num)  # Choose between robbing or skipping
            prev2, prev1 = prev1, new_max  # Update previous values


        return prev1  # Final max amount robbed
```

4.Output:



Problem-4

1. **Aim:** 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 $2 * 10^9$.

2. Objective:

- Learn how to use combinatorial mathematics or dynamic programming for grid-based pathfinding.
- Understand recursive relationships in counting problems.

3.Code:

```
class Solution {
public:

int uniquePaths(int m, int n) {

std::vector<int> aboveRow(n, 1);


for (int row = 1; row < m; row++) {

    std::vector<int> currentRow(n, 1);

    for (int col = 1; col < n; col++) {

        currentRow[col] = currentRow[col - 1] + aboveRow[col];

    }

    aboveRow = currentRow;

}


return aboveRow[n - 1];

    }

};
```

4.Output:

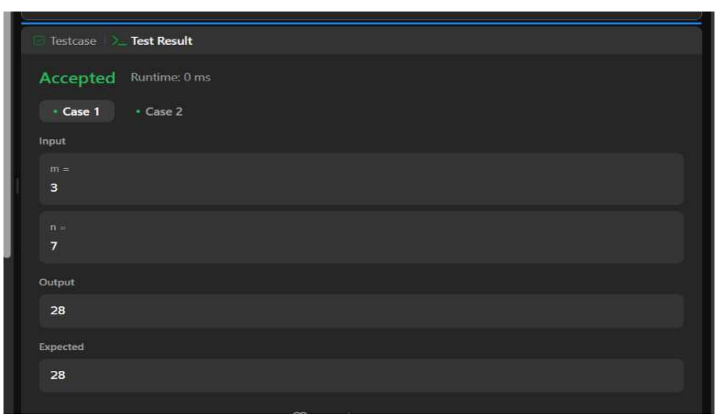


Fig.4:Unique Paths

Problem-5

1. Aim: You are given an integer array nums. You are initially positioned at the array's first index, and 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.

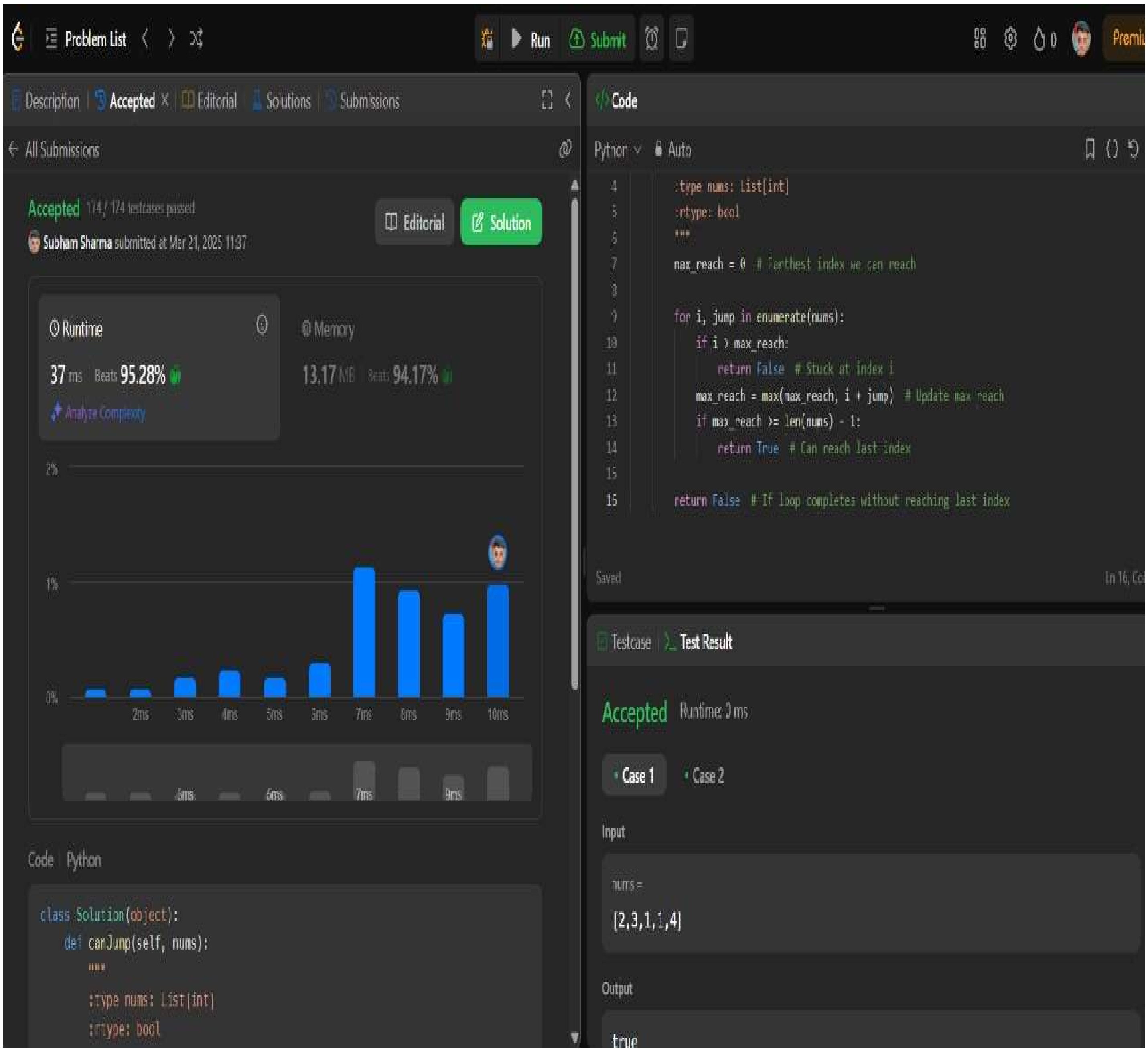
2. Objective:

- Learn dynamic programming approaches to solving the minimum coin change problem.
- Understand the importance of subproblem optimization in making change.

3. Code:

```
class Solution(object):
    def canJump(self, nums):
        """
        :type nums: List[int]
        :rtype: bool
        """
        max_reach = 0  # Farthest index we can reach

        for i, jump in enumerate(nums):
            if i > max_reach:
                return False  # Stuck at index i
            max_reach = max(max_reach, i + jump)  # Update max reach
            if max_reach >= len(nums) - 1:
                return True  # Can reach last index

        return False  # If loop completes without reaching last index
```

4.Output:


Problem List
Run
Submit
Description
Accepted
Editorial
Solutions
Submissions
Code
All Submissions
Python
Auto
Accepted 174 / 174 testcases passed
Subham Sharma submitted at Mar 21, 2025 11:37
Editorial
Solution
Runtime
37 ms | Beats 95.28%
Analyze Complexity
Memory
13.17 MB | Beats 94.17%
:type nums: List[int]
:rtype: bool
max_reach = 0 # Farthest index we can reach
for i, jump in enumerate(nums):
if i > max_reach:
return False # Stuck at index i
max_reach = max(max_reach, i + jump) # Update max reach
if max_reach >= len(nums) - 1:
return True # Can reach last index
return False # If loop completes without reaching last index
Saved
Testcase
Test Result
Accepted Runtime: 0 ms
Case 1
Case 2
Input
nums =
[2,3,1,1,4]
Output
true
Code | Python
class Solution(object):
def canJump(self, nums):
:type nums: List[int]
:rtype: bool

Problem-6

1. **Aim:** Given an integer array nums, return the length of the longest strictly increasing subsequence..

2. **Objective:**

- Learn how to find the longest increasing subsequence using dynamic programming.
- Understand different approaches, including recursive memoization and binary search.

3. **Code:**

```
class Solution {
public:
    int length(vector<int> &nums,vector<int> &dp,int i)
    {
        if(dp[i] != -1)
        {
            return dp[i];
        }
        int res=0,c=0;
        for(int j=i+1;j<nums.size();j++)
        {
            if(nums[j]>nums[i])
            {
                c=1;
                res=max(res,length(nums,dp,j));
            }
        }
        dp[i]=res+1;
        return res+1;
```

```
    }
    int lengthOfLIS(vector<int>& nums) {
        vector<int> dp(nums.size(),-1);
        int res=0;
        for(int j=0;j<nums.size();j++)
        {
            res=max(res,length(nums,dp,j));
        }
        return res;
    }
};
```

4.Output:

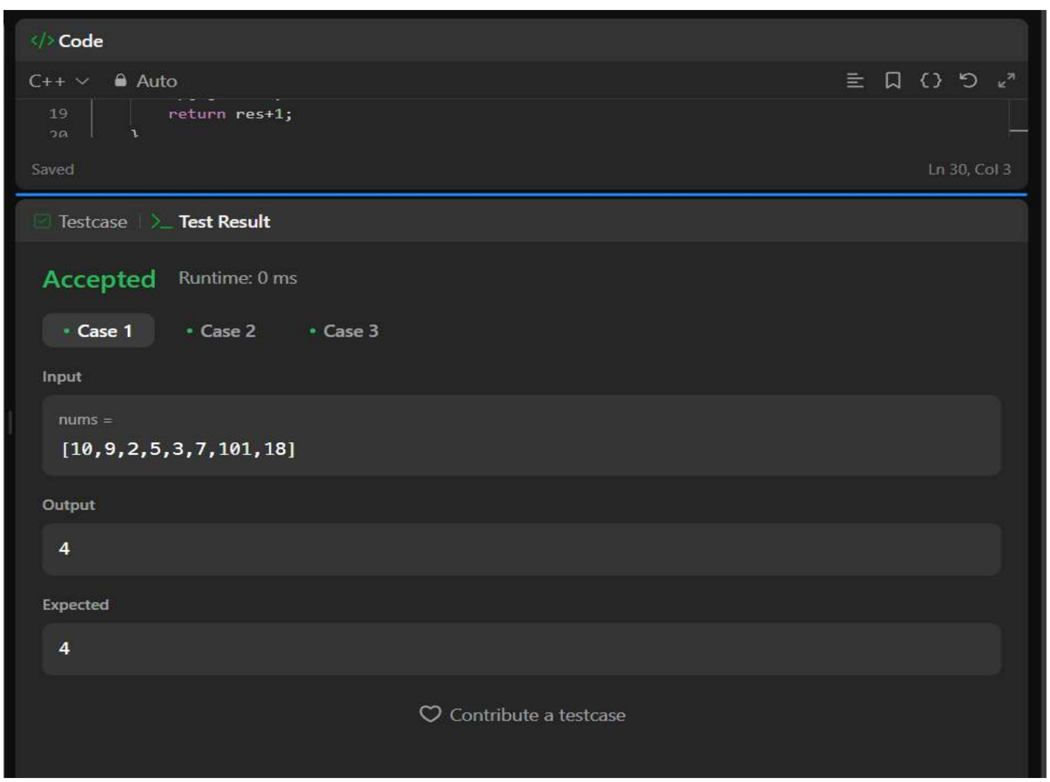


Fig.6:Longest Subsequence

Problem-7

1. Aim: 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.

2. Objective:

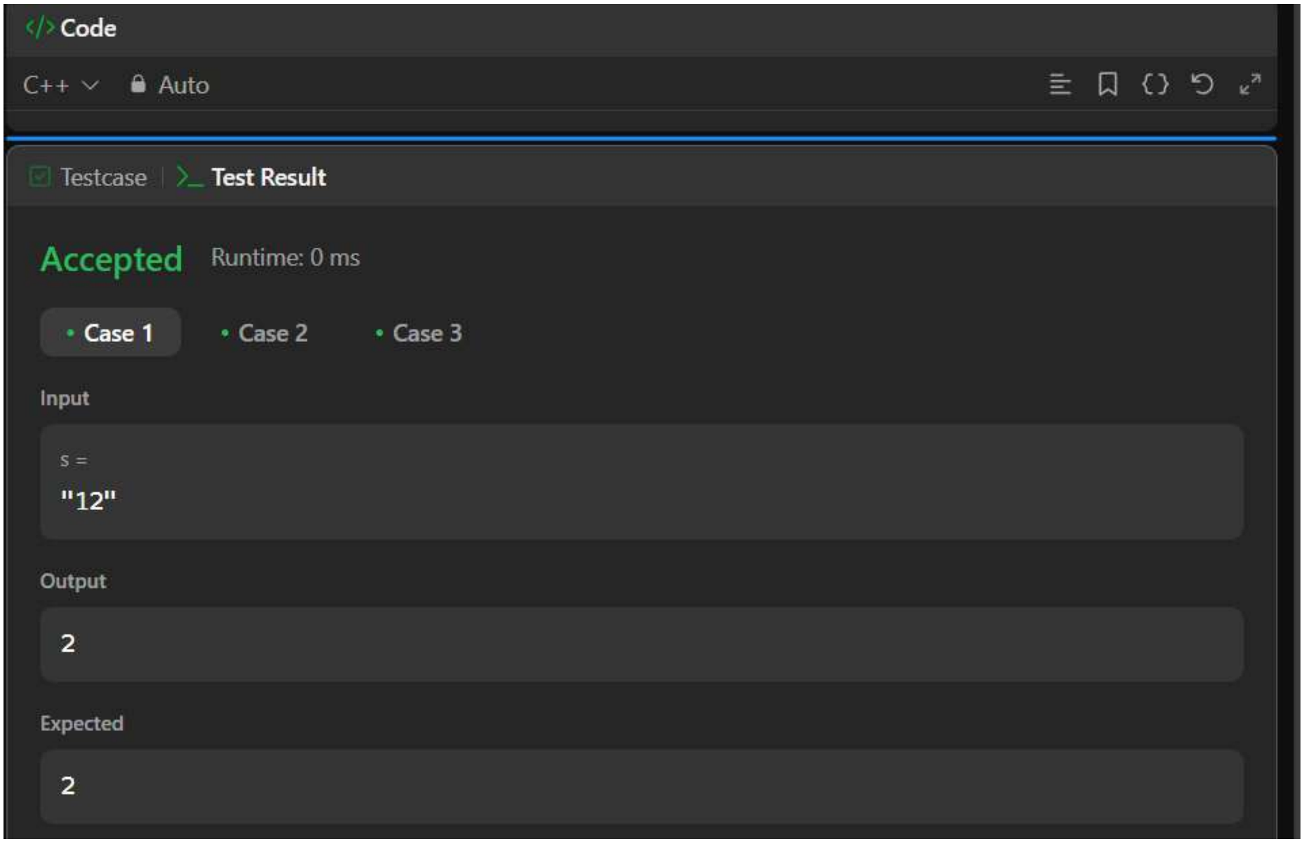
- Learn how to apply dynamic programming to decode messages efficiently.
- Understand constraints on valid numerical character mappings.

3. Code:

```
class Solution {

public:

    int numDecodings(std::string s) {

        if (s.empty() || s[0] == '0') {

            return 0;

        }
```

```
        int n = s.length();
        std::vector<int> dp(n + 1, 0);
        dp[0] = 1;
        dp[1] = 1;

        for (int i = 2; i <= n; ++i) {
            int oneDigit = s[i - 1] - '0';
            int twoDigits = std::stoi(s.substr(i - 2, 2));

            if (oneDigit != 0) {
                dp[i] += dp[i - 1];
            }

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

        return dp[n];
    }
};
```

4.Output:



Problem-8

1. Aim:

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.

2. Objective:

- Learn how to find the minimum number of perfect squares using dynamic programming.
- Understand how to break a number down into square components efficiently.

3.Code:

```
class Solution {
public:

    int numSquares(int n) {

        vector<int> dp(n + 1, INT_MAX);

        dp[0] = 0;

        for (int i = 1; i <= n; ++i) {

            for (int j = 1; j * j <= i; ++j){

            dp[i] = min(dp[i], dp[i - j * j] + 1);

            }

        }

        return dp[n];

    }

};
```

4. Output:

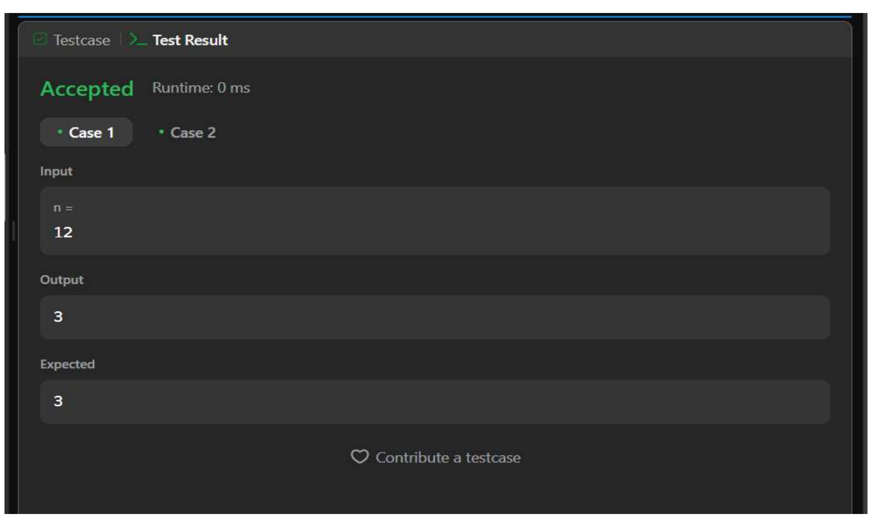


Fig.8: Perfect Squares

Problem-9

1. Aim:

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.

2. Objective:

- Learn how to use dynamic programming to determine if a word can be segmented using a dictionary.
- Understand how to apply set-based lookup for fast verification of substrings.

3. Code:

```
class Solution {
public:
 bool wordBreak(string s, vector<string>& wordDict) {
  const int n = s.length();
  const int maxLength = getMaxLength(wordDict);
  const unordered_set<string> wordSet{begin(wordDict), end(wordDict)};
  vector<int> dp(n + 1);
  dp[0] = true;

  for (int i = 1; i <= n; ++i)
   for (int j = i - 1; j >= 0; --j) {
    if (i - j > maxLength)
    break;
    if (dp[j] && wordSet.count(s.substr(j, i - j))) {
     dp[i] = true;
     break;
    }
```

```
    }
  return dp[n];
 }


private:
 int getMaxLength(const vector<string>& wordDict) {
  return max_element(begin(wordDict), end(wordDict),
                  [](const auto& a, const auto& b) {
                   return a.length() < b.length();
                  })
      ->length();
 }
};
```

4.Output:

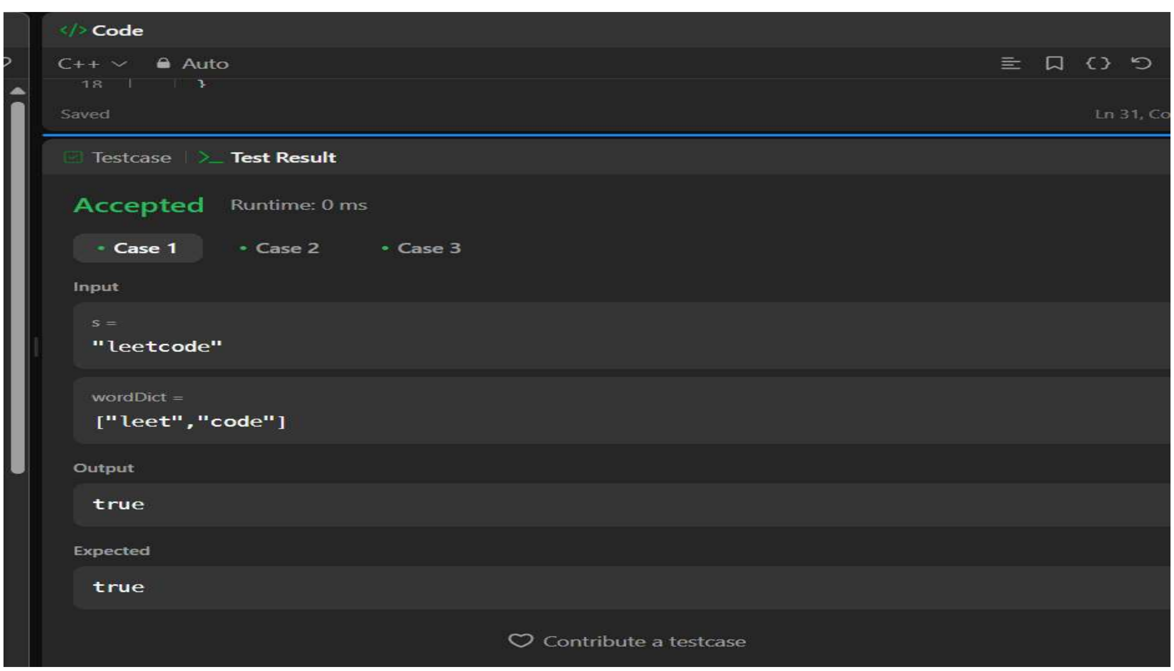


Fig.9: Word Break