**Experiment 7**

**Student Name:** Keshav Mangla **UID:** 22BET10218

**Branch:** BE -IT **Section/Group:**22BET\_IOT-702/A

**Semester:** 6th **Date of Performance:**20/3/2025 **Subject Name:** Advanced Programming Lab-2 **Subject Code:** 22ITP-351

## Climbing Stairs Aim :

To determine the number of distinct ways to climb a staircase with n steps, given that one can take

either 1 or 2 steps at a time.

## Objectives :

* Understand dynamic programming concepts.
* Solve recurrence relation problems efficiently.
* Optimize time complexity from exponential to linear.

**Implementation/Code :**

class Solution {

public int climbStairs(int n) { if (n <= 2) return n;

int first = 1, second = 2; for (int i = 3; i <= n; i++) {

int third = first + second; first = second;

second = third;

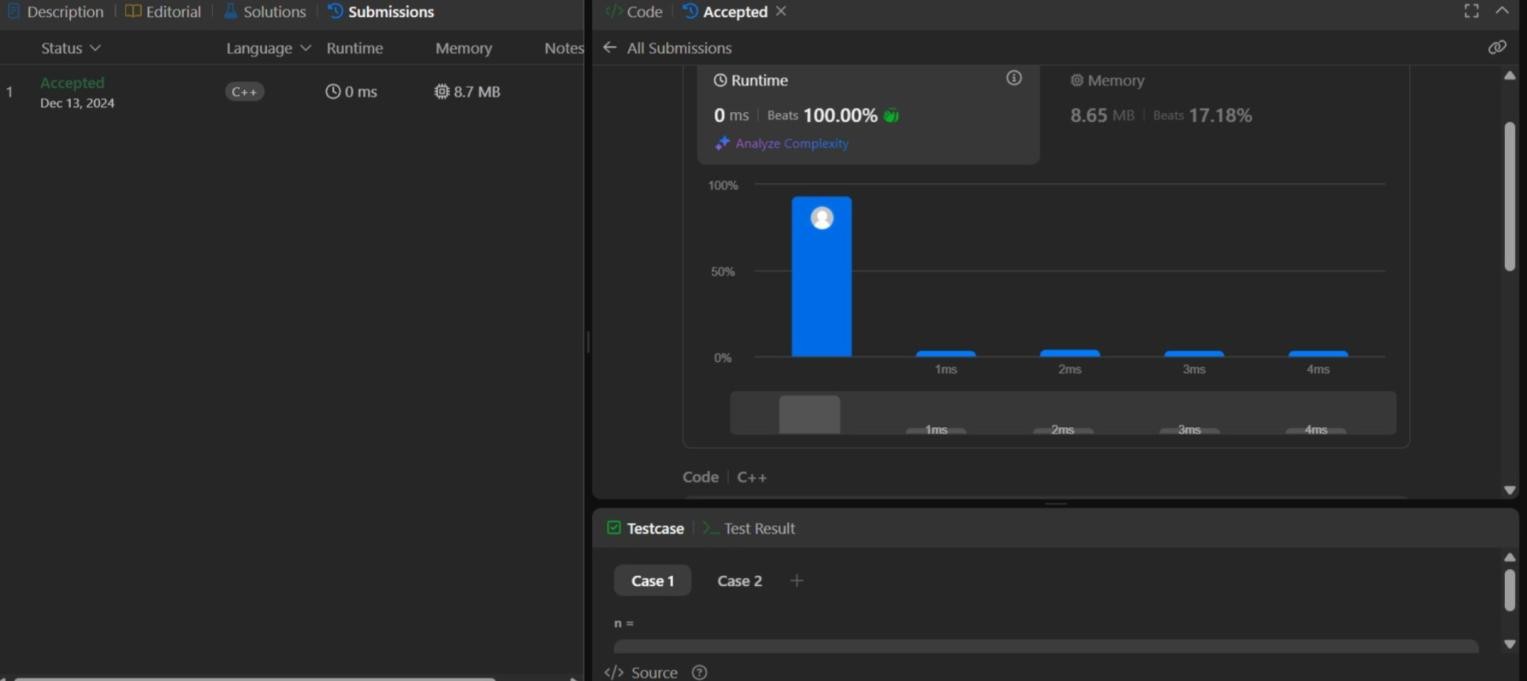
}

return second;

}

}

**Output :**

****

## Best Time to Buy and Sell Stock Aim :

To maximize profit from an array where the i-th element represents the stock price on day i, ensuring that buying happens before selling.

## Objective :

* Implement a greedy approach for stock trading.
* Optimize the time complexity to linear.

## Implementation/Code :

class Solution {

public int maxProfit(int[] prices) {

int minPrice = Integer.MAX\_VALUE; int maxProfit = 0;

for (int price : prices) { if (price < minPrice) {

minPrice = price;

} else {

maxProfit = Math.max(maxProfit, price - minPrice);

}

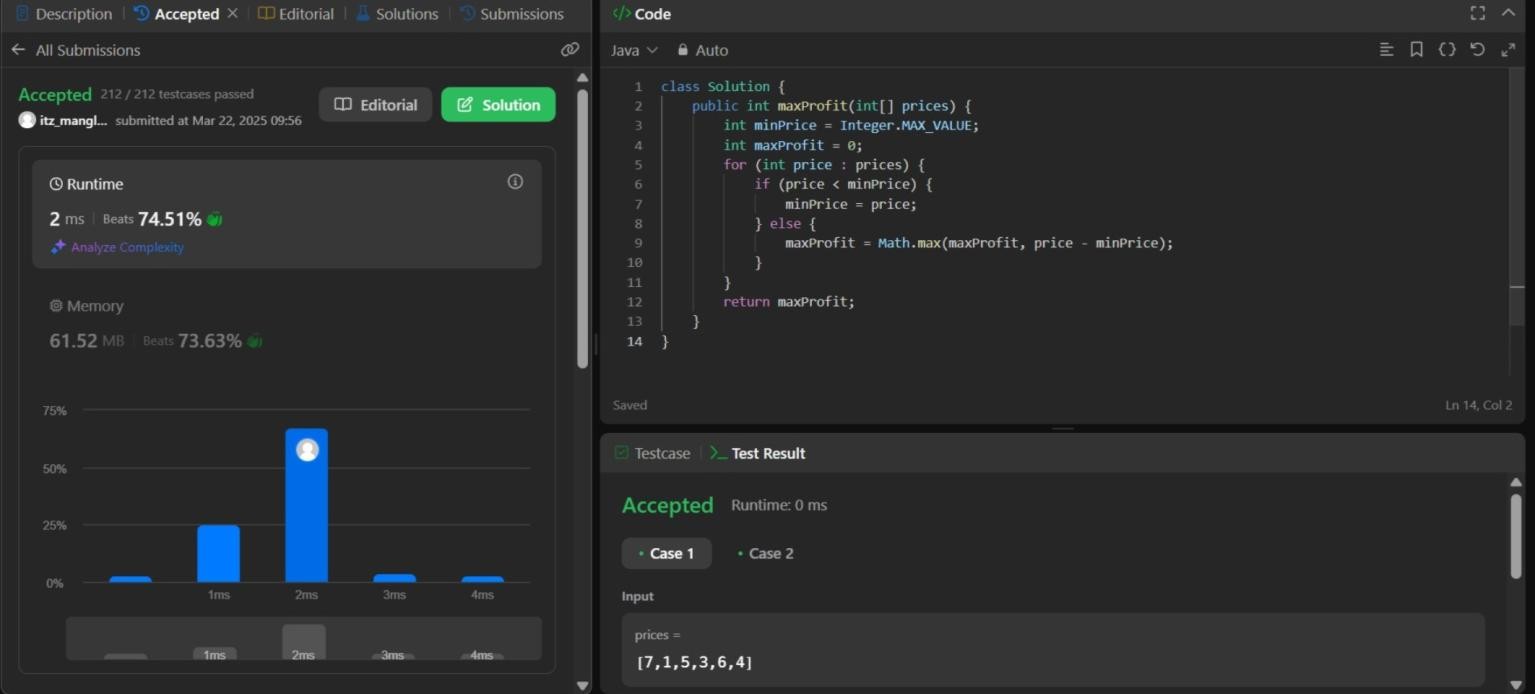
}

return maxProfit;

}

}

**Output :**



**House Robber**

**Aim :**

To determine the maximum amount of money a robber can steal without robbing two adjacent houses.

## Objective :

* Understand dynamic programming in the context of decision-making problems.
* Learn to optimize recursive solutions using memoization.

## Implementation/Code :

class Solution {

public int rob(int[] nums) {

if (nums.length == 0) return 0;

if (nums.length == 1) return nums[0]; int prev1 = 0, prev2 = 0;

for (int num : nums) { int temp = prev1;

prev1 = Math.max(prev2 + num, prev1); prev2 = temp;

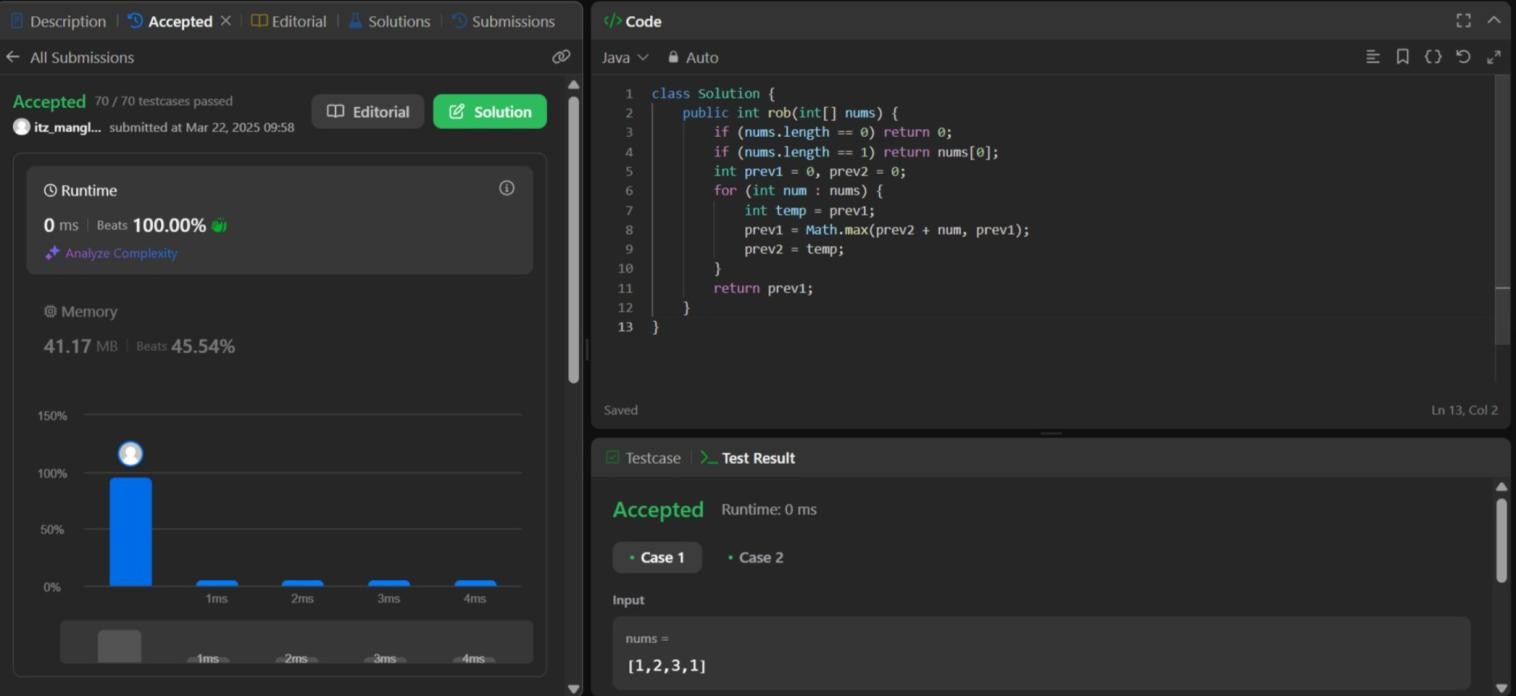
}

return prev1;

}

}

## Output :



**Jump Game Aim :**

To determine whether it is possible to reach the last index from the first index in an array where each element represents the maximum jump length.

## Objective :

* Understand greedy algorithms and dynamic programming applications.
* Optimize backtracking solutions using linear approaches.

## Implementation/Code :

class Solution {

public boolean canJump(int[] nums) { int maxReach = 0;

for (int i = 0; i < nums.length; i++) { if (i > maxReach) return false;

maxReach = Math.max(maxReach, i + nums[i]);

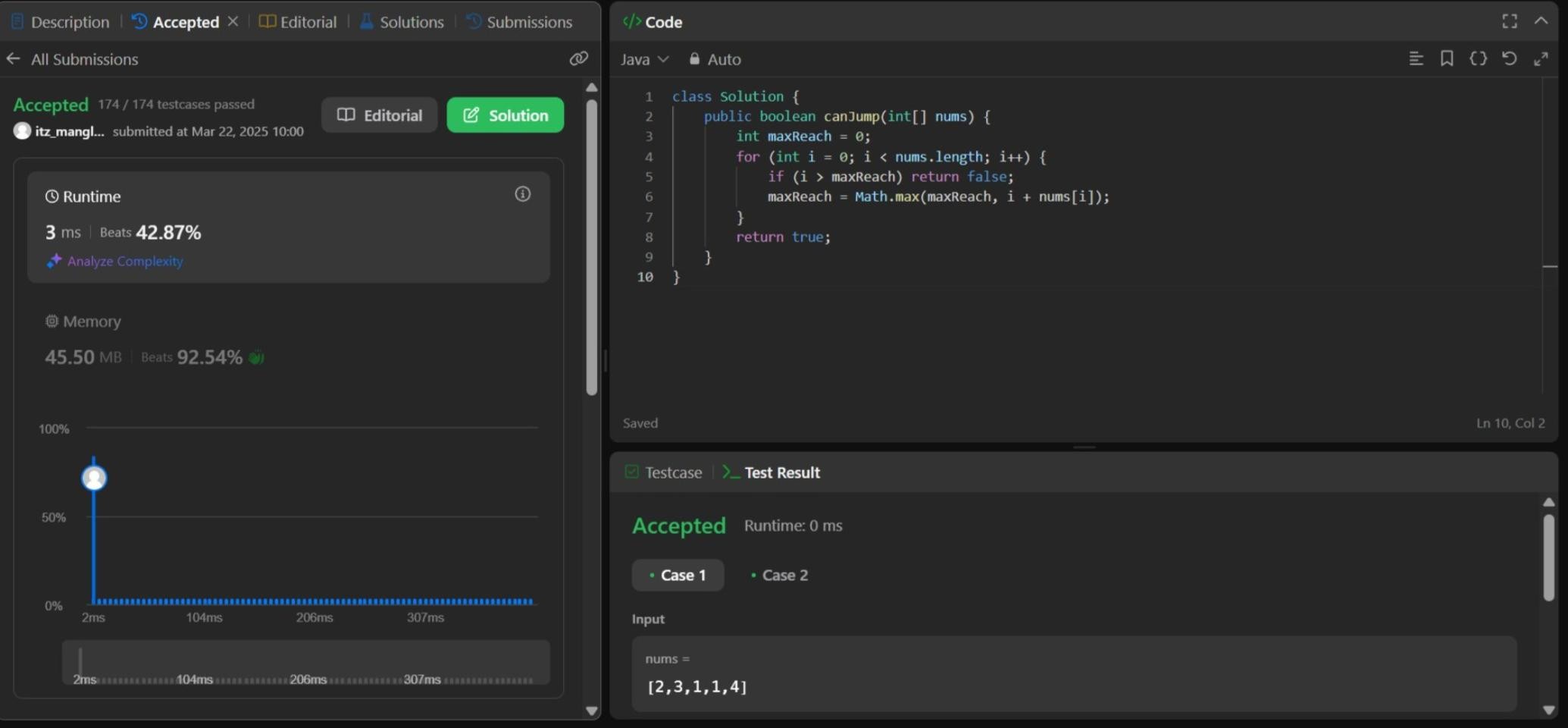
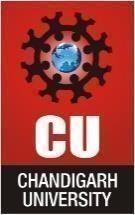
}

return true;

}

}

**Output :**



COMPUTER SCIENCE & ENGINEERING



# & ENGINEERING



COMPUTER SCIENCE

## Maximum Product Subarray Aim :

To find the contiguous subarray within an array that has the largest product.

## Objectives :

* Handle both positive and negative values in arrays.
* Learn how to track maximum and minimum products simultaneously.

## Implementation/Code :

class Solution {

public int maxProduct(int[] nums) {

int maxProduct = nums[0], minProduct = nums[0], result = nums[0]; for (int i = 1; i < nums.length; i++) {

if (nums[i] < 0) {

int temp = maxProduct; maxProduct = minProduct; minProduct = temp;

}

maxProduct = Math.max(nums[i], maxProduct \* nums[i]); minProduct = Math.min(nums[i], minProduct \* nums[i]); result = Math.max(result, maxProduct);

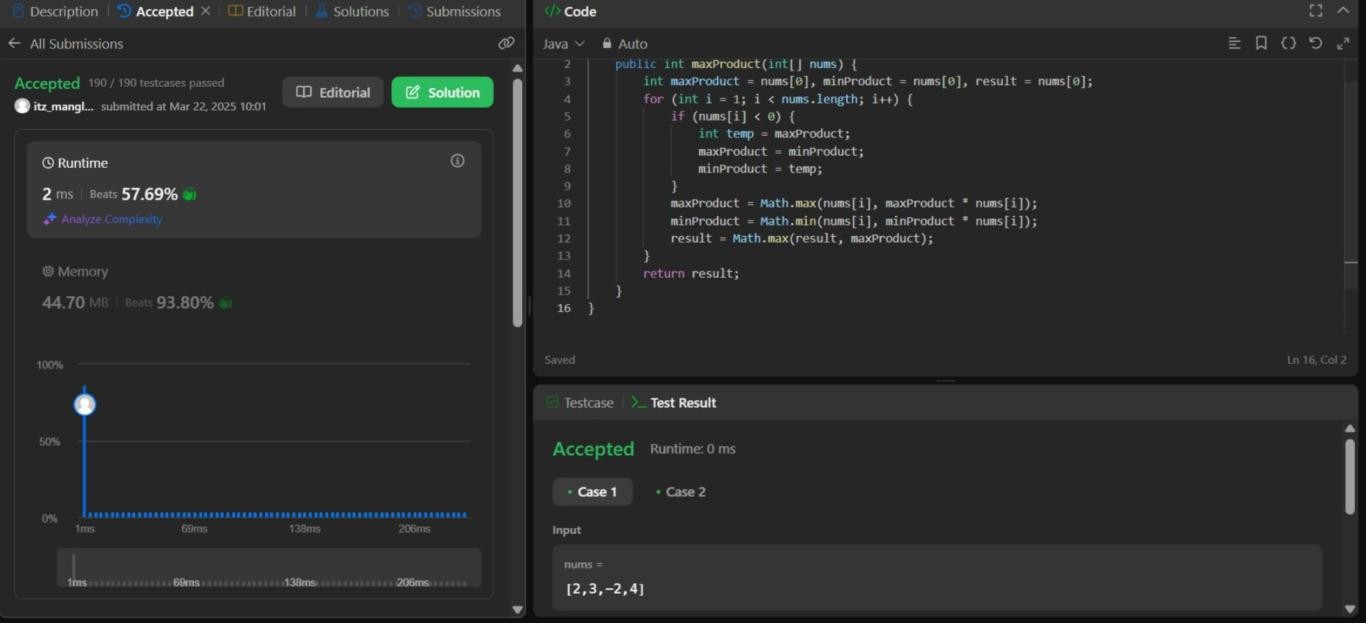
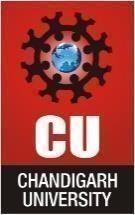
}

return result;

}

}

**Output :**



COMPUTER SCIENCE & ENGINEERING





COMPUTER SCIENCE

**Perfect Squares**

**Aim :**

To find the minimum number of perfect square numbers that sum to a given n.

## Objectives :

* Implement dynamic programming for optimization.
* Explore mathematical approaches for problem-solving.

## Code :

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

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

}

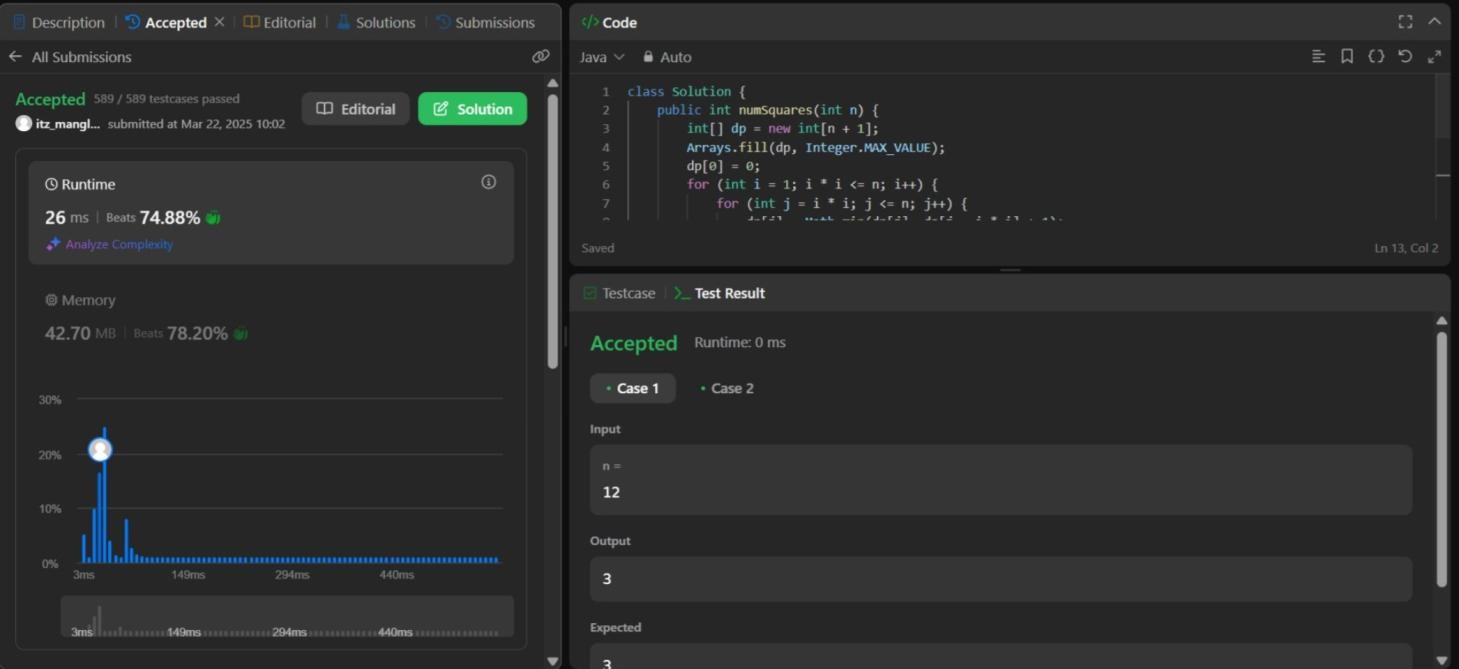
}

return dp[n];

}

}

## Output:



**Coin Change Aim :**

To determine the minimum number of coins needed to make up a given amount.

## Objectives :

* Explore dynamic programming approaches for minimizing computations.
* Implement efficient bottom-up solutions.

## 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 j = coin; j <= amount; j++) {

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

}

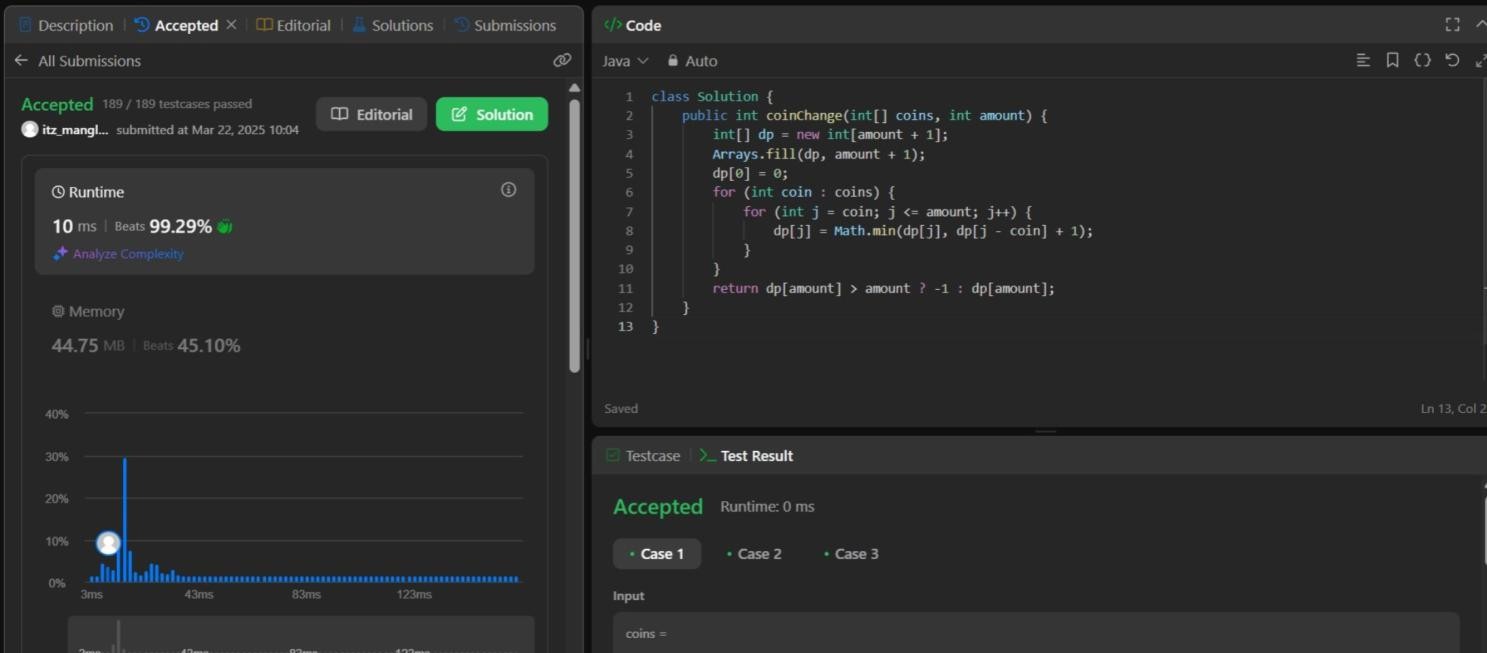
}

return dp[amount] > amount ? -1 : dp[amount];

}

}

## Output :



**Learning Outcomes :**

* Proficiency in solving optimization problems.
* Understanding of dynamic programming and coin denomination strategies.
* Mastery of dynamic programming techniques.
* Understanding mathematical properties of numbers.
* Understanding of how negative numbers affect product calculations.
* Efficient approach using dynamic programming.