Experiment-7

Student Name: Anshuman Raj **UID:** 22BET10081
Branch: BE-IT **Section:** 22BET_IOT-702 ‘A’
Semester: 6th **Date of Performance:**20/03/25
Sub Name: Advanced Programming Lab-2 **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:

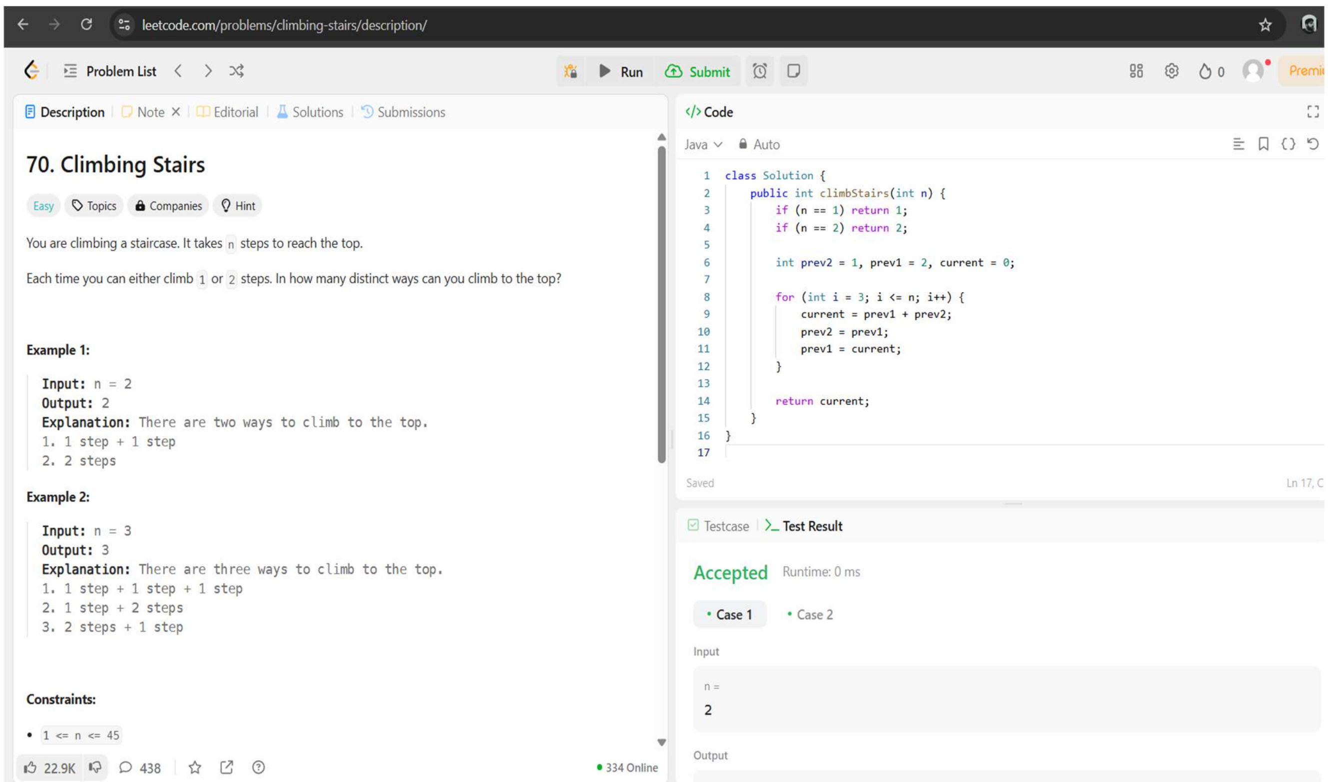
1. Find the number of distinct ways to reach the nth step when you can climb 1 or 2 steps at a time.
2. Use dynamic programming or mathematical approaches to optimize computation.
3. Implement an efficient solution with minimal time and space complexity.
4. Explore different approaches like Recursion, DP, Iteration, and Matrix Exponentiation for performance trade-offs.

3. Code:

```
class Solution {
public int climbStairs(int n) {
if (n == 1) return 1;
if (n == 2) return 2;
int prev2 = 1, prev1 = 2, current = 0;
for (int i = 3; i <= n; i++) {
current = prev1 + prev2;
prev2 = prev1;
prev1 = current;
```

```
}
return current;
}
```

4. Output:



Climbing the stairs

5. Learning Outcomes:

1. Understanding Dynamic Programming and how to optimize recursion.
2. Recognizing the Fibonacci sequence pattern in real-world problems.
3. Implementing space-optimized DP to reduce memory usage.
4. Exploring efficient mathematical techniques like matrix exponentiation for faster computation.

Problem 2

1. Aim:

Write a program for best time to buy and sell stock.

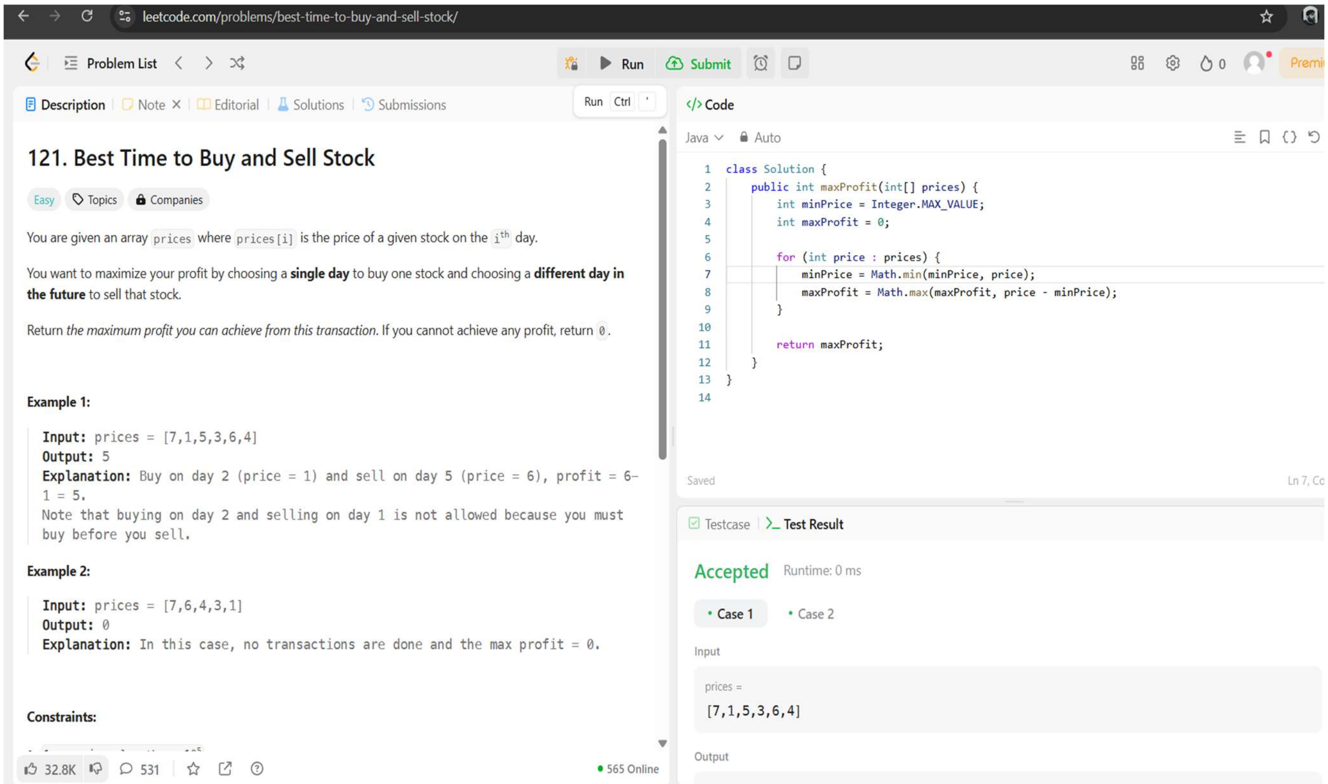
2. Objective:

1. Maximize profit by buying and selling a stock once.
2. Ensure the buy happens before the sell in the given price list.
3. Optimize the solution to run in O(n) time complexity with O(1) space.
4. Utilize dynamic tracking of the minimum price and maximum profit.

3.Code:

```
class Solution {

public int maxProfit(int[] prices) {

int minPrice = Integer.MAX_VALUE;

int maxProfit = 0;

for (int price : prices) {

minPrice = Math.min(minPrice, price);

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

}

return maxProfit;

}

}
```

4.Output:



Buy & sell books

5.Learning Outcomes:

1. Greedy approach for optimal decision-making.
2. Tracking min/max values dynamically in a single pass.
3. Efficient array traversal without extra space.
4. Real-world application of stock trading logic in programming.

Problem 3

1.Aim:

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.

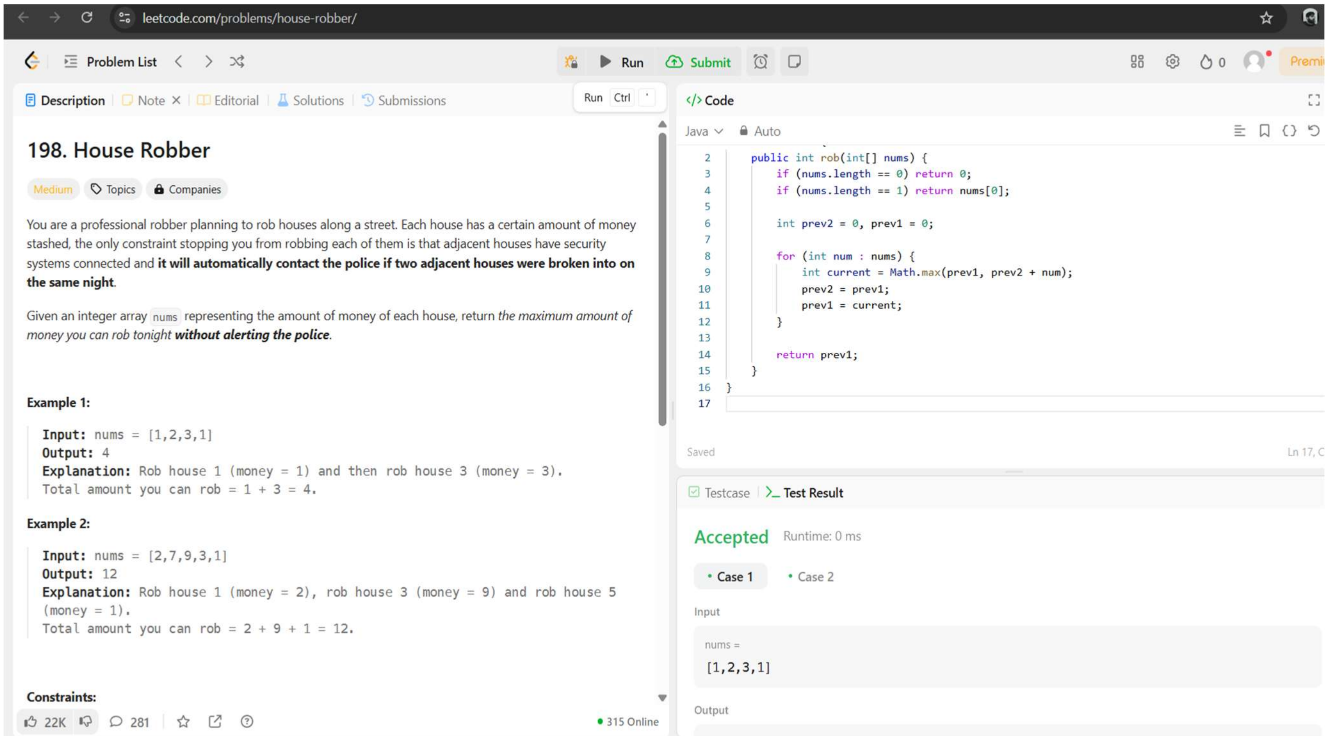
2.Objective:

1. Maximize the amount of money robbed without robbing two adjacent houses.
2. Use dynamic programming to track the optimal subproblems.
3. Achieve O(n) time complexity and O(1) space optimization if possible.
4. Utilize bottom-up DP approach or memoization to improve efficiency.

3.Code:

```
class Solution {
public int rob(int[] nums) {
if (nums.length == 0) return 0;
if (nums.length == 1) return nums[0];
int prev2 = 0, prev1 = 0;
for (int num : nums) {
int current = Math.max(prev1, prev2 + num);
prev2 = prev1;
prev1 = current;
}
return prev1;
}
}
```

4.Output:



House Robber

5.Learning Outcomes:

1. Understanding dynamic programming for optimization.
2. Learning to handle constraints efficiently (avoiding adjacent selections).
3. Implementing state transition logic dynamically in a single pass.
4. Applying real-world security logic in algorithmic thinking.

Problem 4

1.Aim:

Write a program to decode the string in different ways.

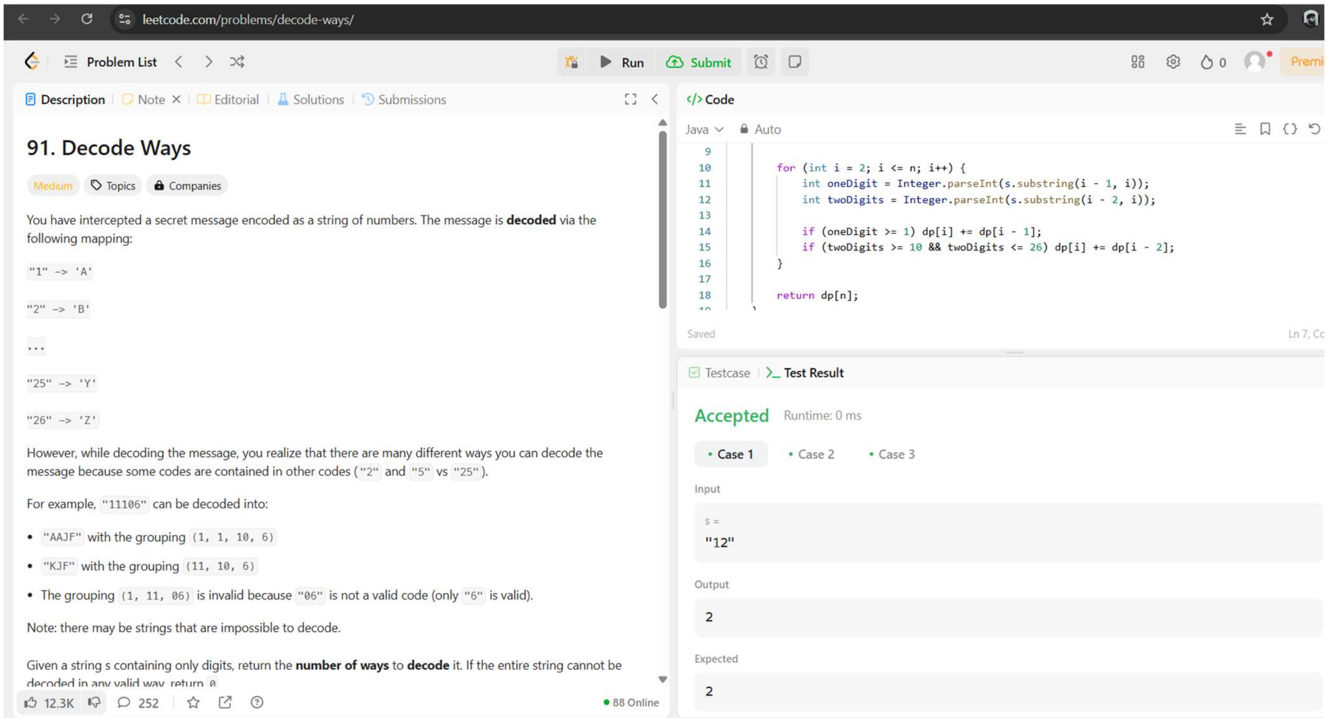
2.Objective:

1. Count the number of ways to decode a given numeric string where:
2. '1' → 'A', '2' → 'B', ..., '26' → 'Z'.
3. Handle leading zeros and invalid sequences correctly.
4. Use Dynamic Programming (DP) to optimize the solution in O(n) time.
5. Implement bottom-up DP or recursive memoization to avoid recomputation.

3.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];
}
}
```

4.Output:



Decode Ways

5.Learning Outcomes:

1. Mastering Dynamic Programming for sequence-based problems.
2. Handling leading zeros and invalid cases correctly.
3. Implementing state transitions efficiently to minimize computation.
4. Understanding real-world encoding/decoding applications in software.

Problem 5

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.

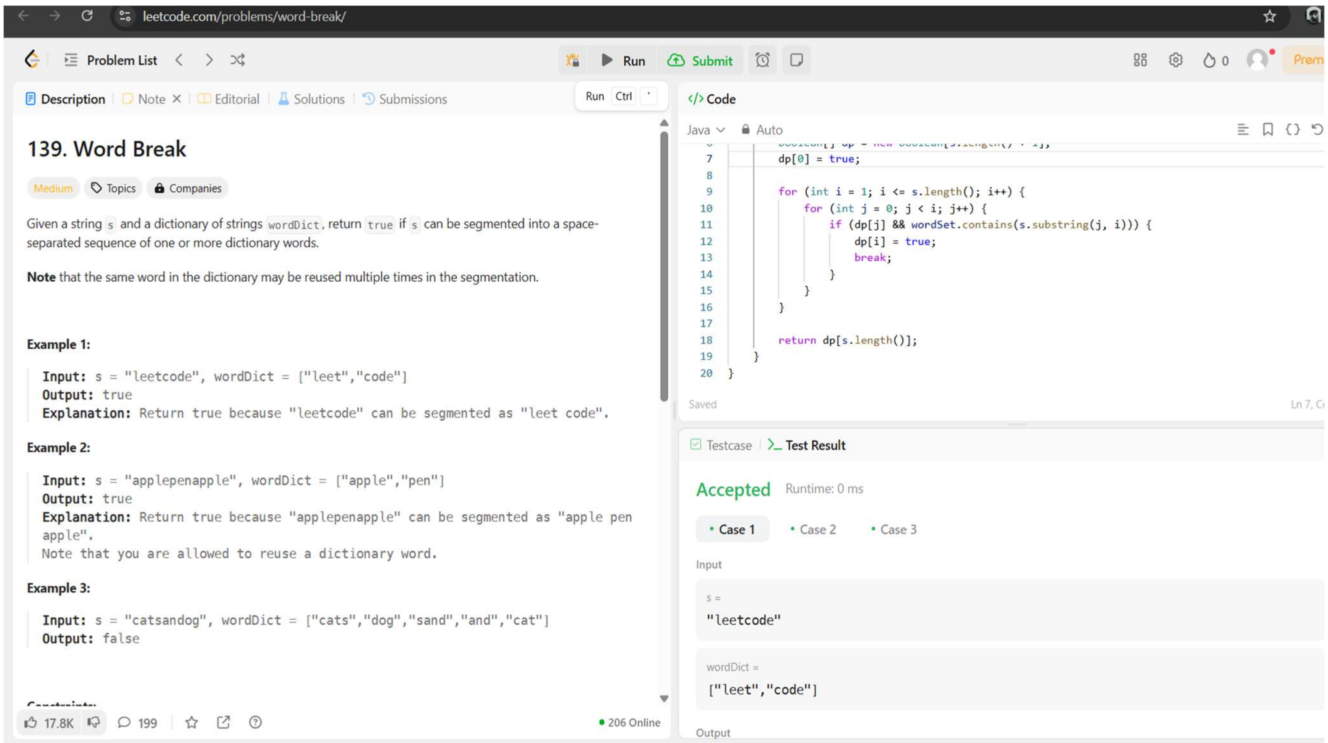
2.Objective:

1. Determine if a given string can be segmented into words from a dictionary.
2. Use Dynamic Programming (DP) to efficiently check valid partitions.
3. Optimize the solution to run in $O(n^2)$ time complexity.
4. Utilize a HashSet for fast lookups and bottom-up DP for efficiency.

3.Code:

```
import java.util.*;
class Solution {
public boolean wordBreak(String s, List<String> wordDict) {
Set<String> wordSet = new HashSet<>(wordDict);
boolean[] dp = new boolean[s.length() + 1];
dp[0] = true; // Base case: empty string is valid
for (int i = 1; i <= s.length(); i++) {
for (int j = 0; j < i; j++) {
if (dp[j] && wordSet.contains(s.substring(j, i))) {
dp[i] = true;
break;
}
}
}
return dp[s.length()];
}
}
```

4.Output:



Word Break

5.Learning Outcomes:

1. Mastering Dynamic Programming for string segmentation problems.
2. Efficient use of HashSet for fast word lookups.
3. Understanding substring processing and nested iteration optimization.
4. Applying real-world NLP techniques for text segmentation.

Problem 6

1.Aim:

Given an integer n, return the least number of perfect square numbers that sum to n.

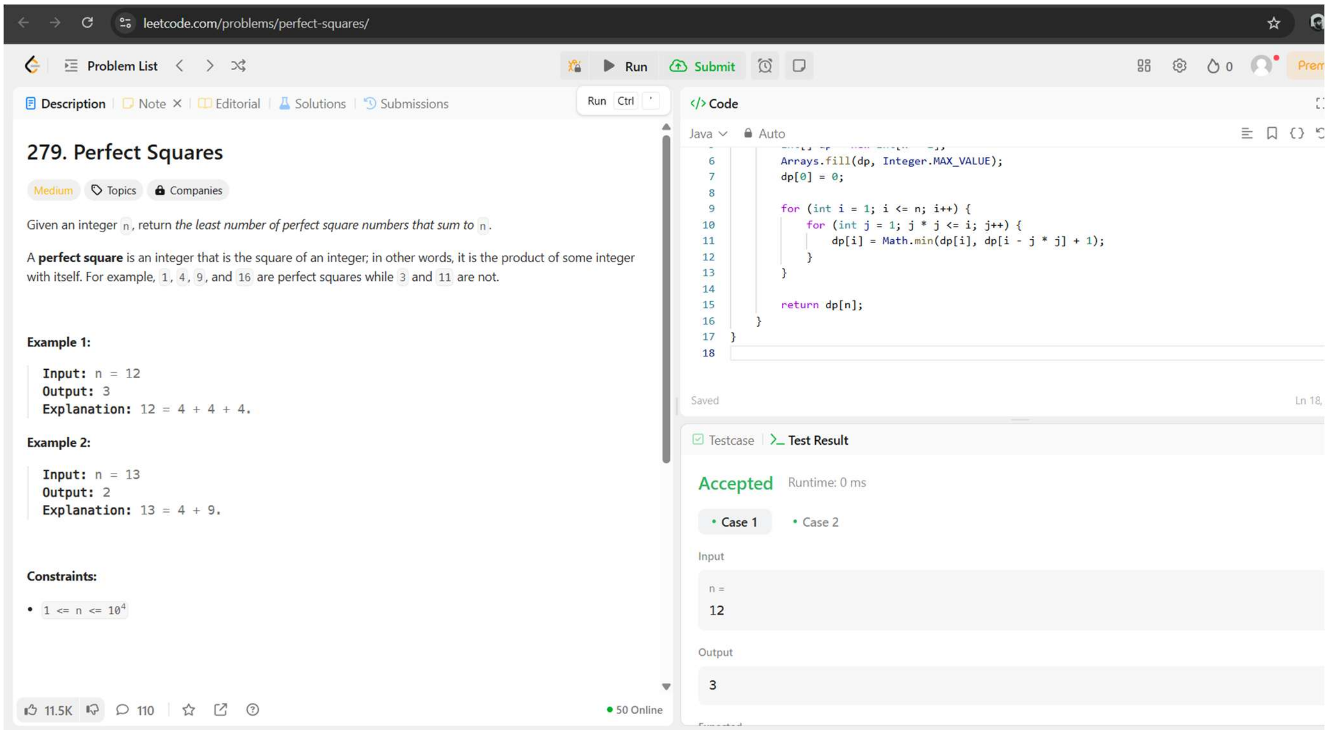
2.Objective:

1. Find the minimum number of perfect squares (e.g., 1, 4, 9, 16, ...) that sum up to n.
2. Use Dynamic Programming (DP) or BFS for an optimal solution.
3. Achieve $O(n\sqrt{n})$ time complexity using DP or $O(\sqrt{n})$ using Lagrange's Four-Square Theorem.
4. Implement an efficient approach avoiding redundant calculations.

3.Code:

```
import java.util.*;
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];
}
}
```

4.Output:



Perfect Square

5.Learning Outcomes:

1. Understanding DP for partitioning problems (breaking n into squares).
2. Optimizing nested loops for performance.
3. Exploring alternative approaches like BFS and Mathematical methods.
4. Applying graph traversal (BFS) for shortest path problems.

Problem 7

1.Aim:

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.

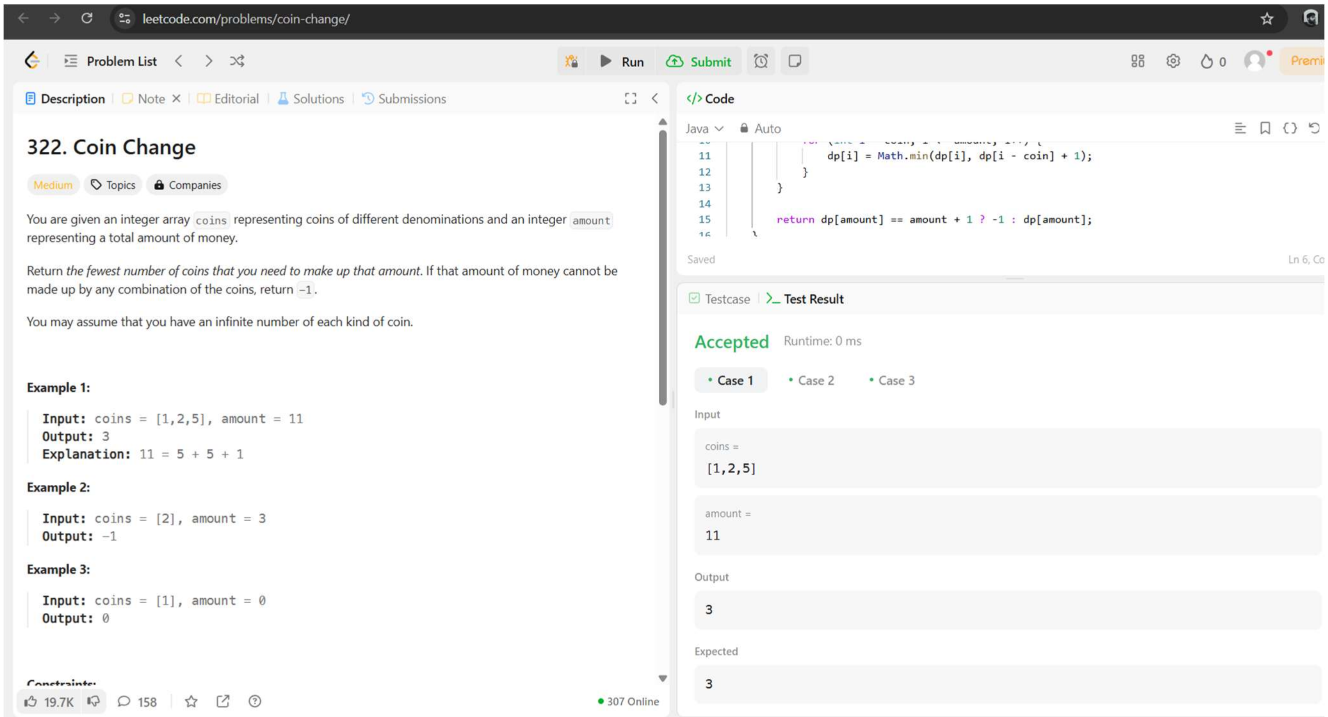
2.Objective:

1. Find the minimum number of coins needed to make a given amount.
2. Use Dynamic Programming (DP) to optimize the solution efficiently.
3. Achieve O(n * m) time complexity using bottom-up DP, where n is the amount and m is the number of coins.
4. Implement an iterative DP approach or BFS for shortest path optimization.

3.Code:

```
import java.util.*;
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];
}
}
```

4.Output:



Coin Change

5.Learning Outcomes:

1. Mastering Dynamic Programming for minimization problems.
2. Efficiently handling unbounded knapsack-like problems.
3. Implementing bottom-up DP with state transitions.
4. Exploring alternative solutions like BFS for shortest path computation.