



DEPARTMENT OF COMPUTERSCIENCE & ENGINEERING

Discover. Learn. Empower.

Experiment-7

Name: Sushil Kumar

Branch: BE-IT

Semester: 6th

Subject Name: AP LAB-II

UID: 22BET10343

Section/Group: 22BET_IOT-702/A

Date of Performance: 20/03/25

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 {  
  
public:  
  
    int climbStairs(int n) {  
  
        if (n == 0 || n == 1) {  
  
            return 1;  
  
        }  
  
        vector<int> dp(n+1);  
  
        dp[0] = dp[1] = 1;  
  
        for (int i = 2; i <= n; i++) {  
  
            dp[i] = dp[i-1] + dp[i-2];  
  
        }  
    }  
};
```

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

4.Output:

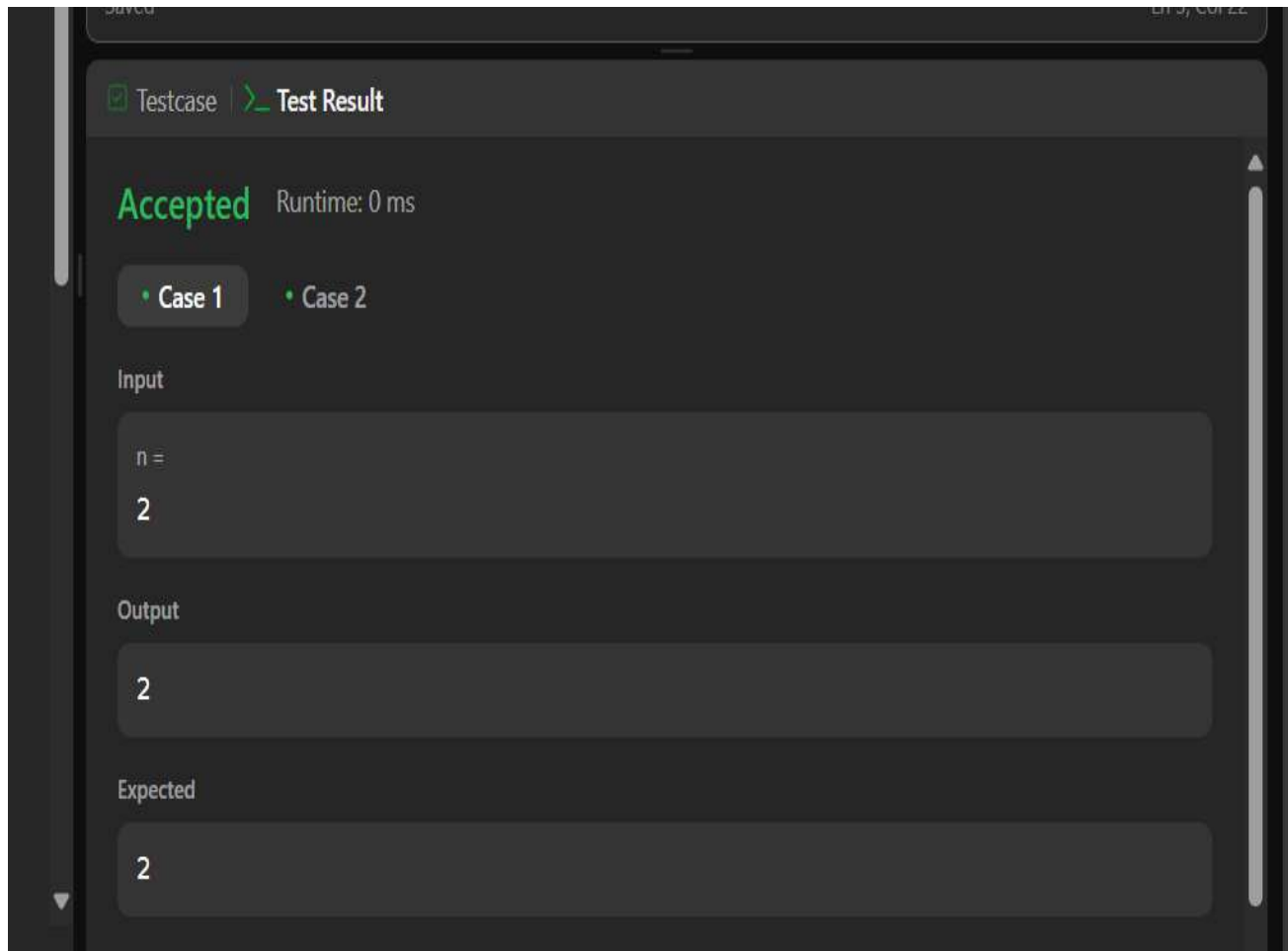


Fig.1:Climbing Stairs

Problem-2

1. Aim:

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

2. Objective:

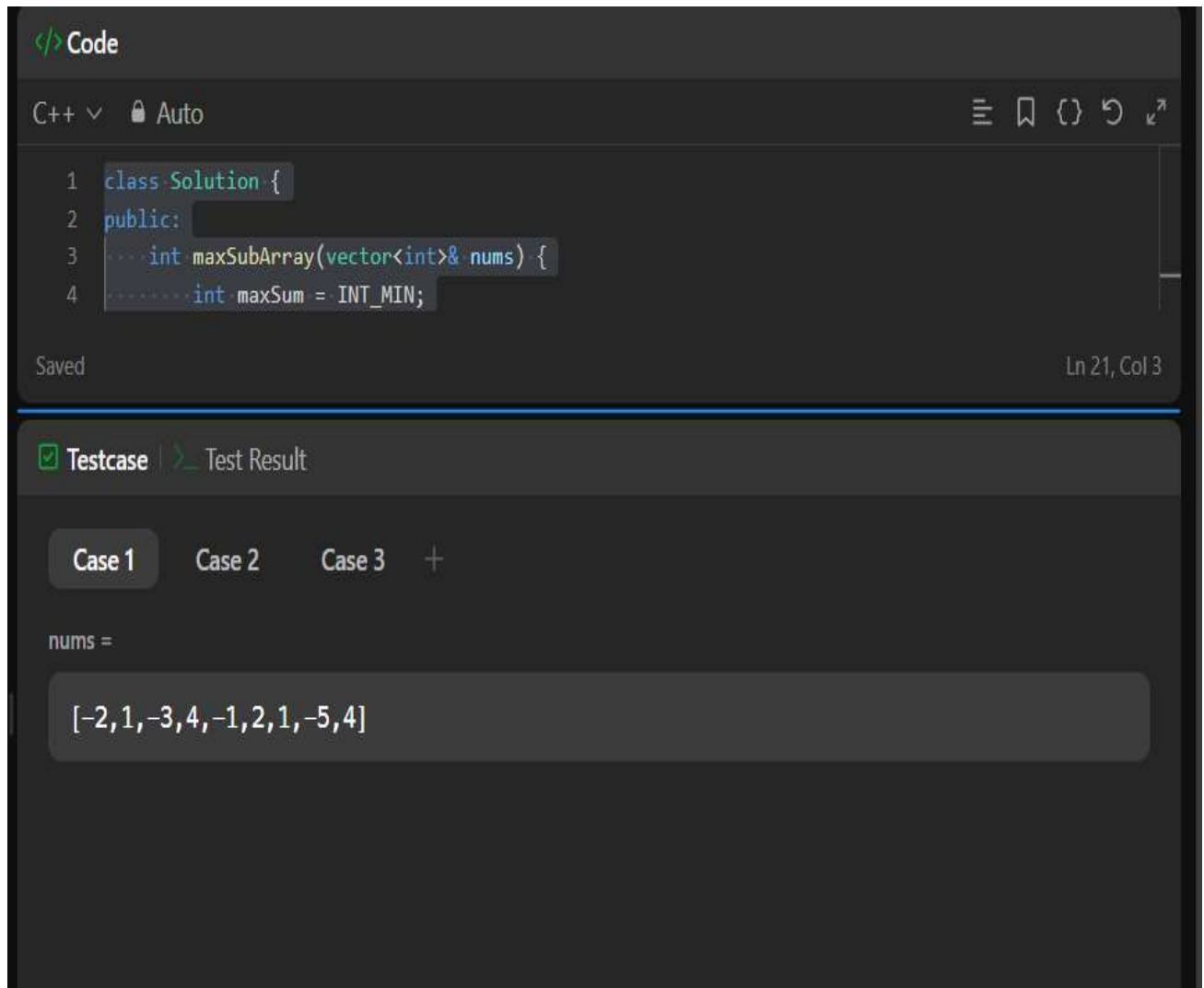
- Learn how to apply Kadane's algorithm to find the maximum subarray sum.
- Understand the concept of dynamic programming in optimizing subarray problems.

3. Code:

```
class Solution {  
  
public:  
  
    int maxSubArray(vector<int>& nums) {  
        int maxSum = INT_MIN;  
        int currentSum = 0;  
  
        for (int i = 0; i < nums.size(); i++) {  
            currentSum += nums[i];  
  
            if (currentSum > maxSum) {  
                maxSum = currentSum;  
            }  
  
            if (currentSum < 0) {  
                currentSum = 0;  
            }  
        }  
    }  
}
```

```
        return maxSum;  
    }  
};
```

4.Output:



The screenshot shows a C++ IDE with a dark theme. The top panel is titled 'Code' and shows the following code:

```
1 class Solution {  
2 public:  
3     int maxSubArray(vector<int>& nums) {  
4         int maxSum = INT_MIN;
```

The bottom panel is titled 'Testcase' and shows a test case for 'Case 1'. The input is 'nums = [-2,1,-3,4,-1,2,1,-5,4]'.

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.

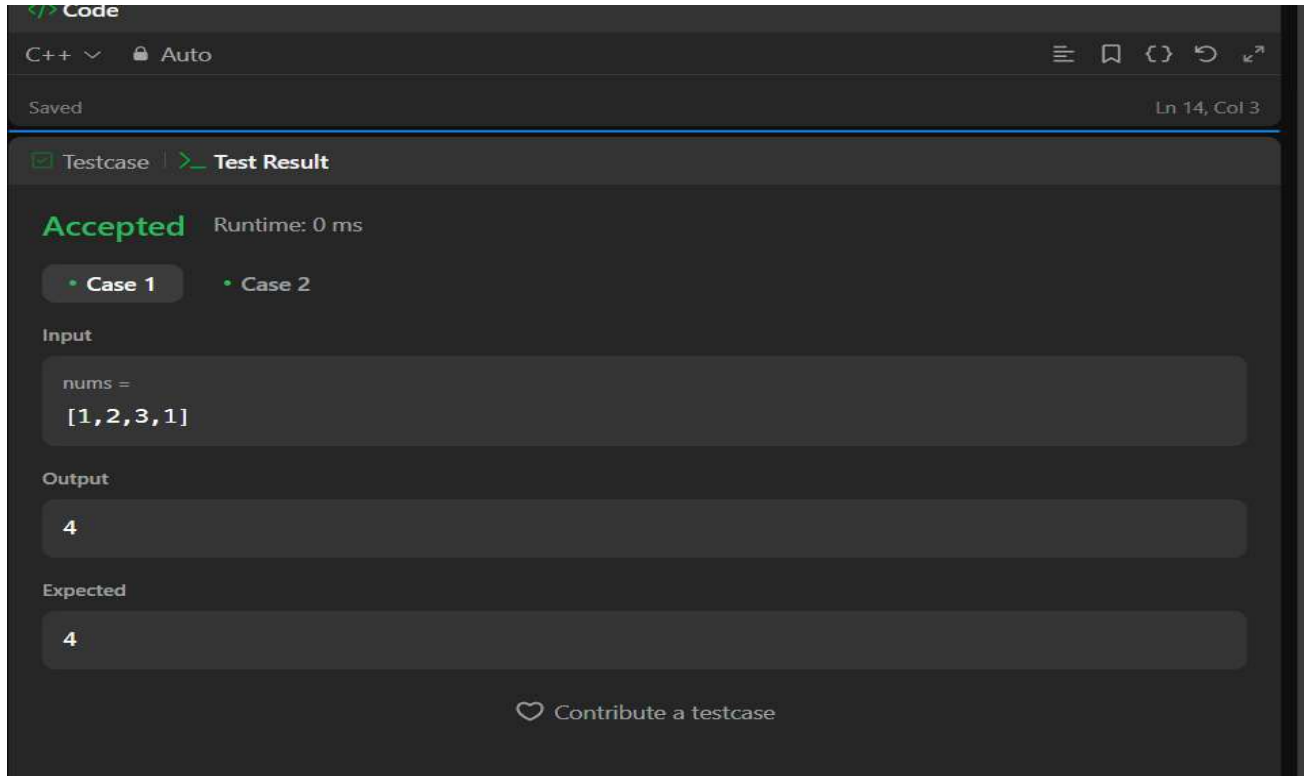
Given an integer array `nums` representing the amount of money of each house, return the maximum amount of money you can rob tonight without alerting the police.

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 {  
  
public:  
  
    int rob(vector<int>& nums) {  
  
        int rob = 0;  
  
        int norob = 0;  
  
        for (int i = 0; i < nums.size(); i++) {  
  
            int newRob = norob + nums[i];  
  
            int newNoRob = max(norob, rob);  
  
            rob = newRob;  
  
            norob = newNoRob;  
  
        }  
  
        return max(rob, norob);  
  
    }  
  
};
```



The screenshot shows a code editor interface with a dark theme. At the top, it says 'Code' and 'C++' with a dropdown arrow. Below that, it says 'Auto' and 'Saved'. On the right, it says 'Ln 14, Col 3'. The main area is divided into two tabs: 'Testcase' and 'Test Result'. The 'Test Result' tab is active, showing 'Accepted' in green text and 'Runtime: 0 ms'. Below this, there are two buttons: 'Case 1' and 'Case 2'. Under 'Case 1', there is an 'Input' section with a text box containing 'nums = [1,2,3,1]'. Below that is an 'Output' section with a text box containing '4'. At the bottom, there is an 'Expected' section with a text box containing '4'. At the very bottom, there is a link that says 'Contribute a testcase' with a heart icon.

4.Output:

Fig.3: House Robber

Problem-4

1.Aim:There is a robot on an $m \times n$ grid. The robot is initially located at the top-left corner (i.e., $\text{grid}[0][0]$). The robot tries to move to the bottom-right corner (i.e., $\text{grid}[m - 1][n - 1]$). 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:

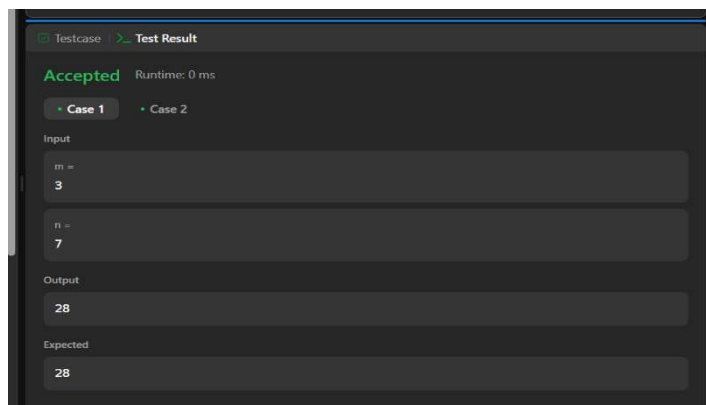


Fig.4:Unique Paths



Problem-5

1.Aim: You are given an integer array 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.

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 {  
  
public:  
  
    int minCoin(vector<int> &coins, int amount) {  
  
        if (amount <= 0) {  
  
            return 0;  
  
        }  
  
  
  
        int ans = INT_MAX;  
        for (int i = 0; i < coins.size(); i++) {  
  
            int coin = coins[i];  
  
            if (coin <= amount) {  
  
                int recAns = minCoin(coins, amount - coin);  
  
  
                if (recAns != INT_MAX) {  
  
                    recAns += 1;  
  
                    ans = min(ans, recAns);  
  
                }  
            }  
        }  
    }  
};
```



```
}  
  
}  
  
return ans;  
  
}  
  
int coinChange(vector<int>& coins, int amount) {  
    int ans = minCoin(coins, amount);  
    if (ans == INT_MAX) {  
        return -1;  
    }  
    return ans;  
}  
};
```

4.Output:

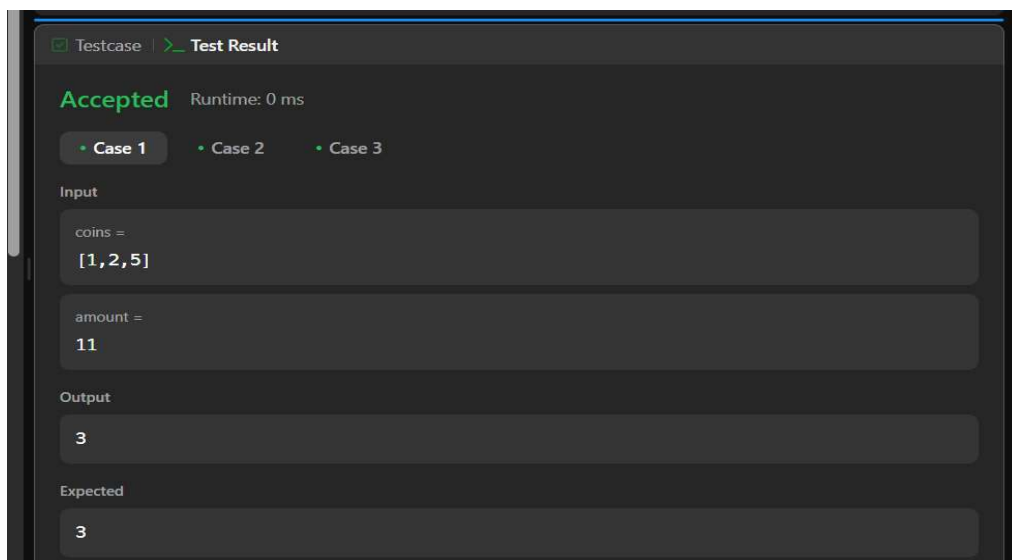


Fig.5: Coin Change



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));
```

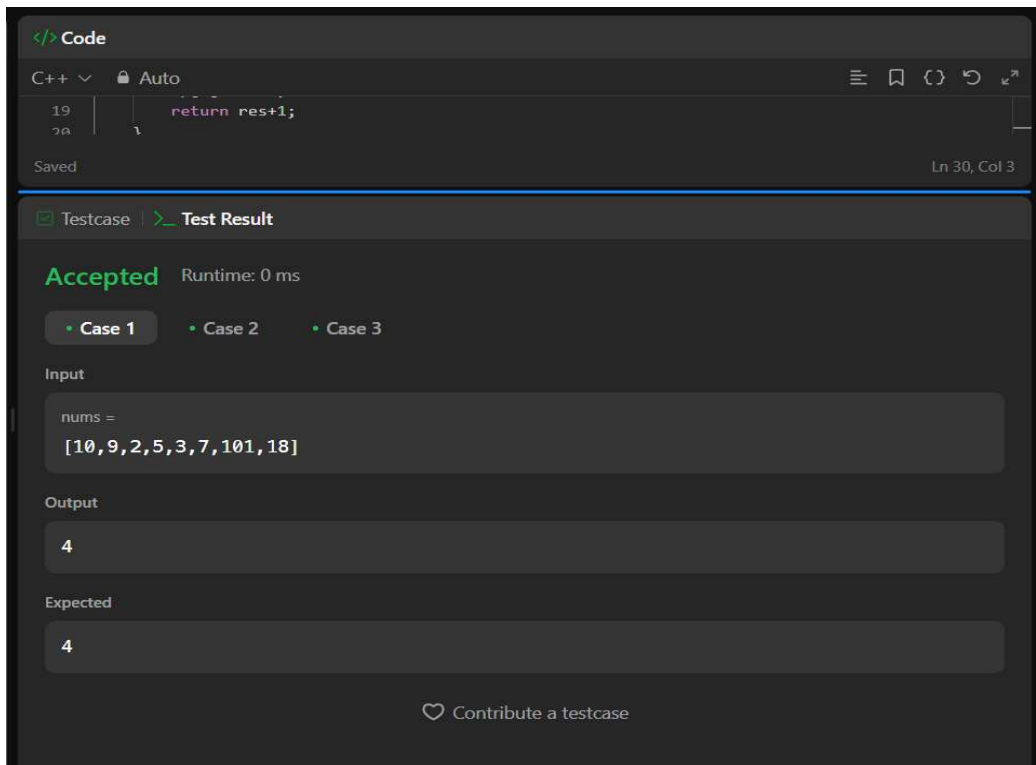
```
    }
```

```
    return res;
```

```
}
```

```
};
```

4.Output:



```
</> Code
C++ v Auto
19 return res+1;
20
Saved Ln 30, Col 3

Testcase Test Result
Accepted Runtime: 0 ms
• Case 1 • Case 2 • Case 3
Input
nums =
[10,9,2,5,3,7,101,18]
Output
4
Expected
4
Contribute a testcase
```

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;  
        }  
    }
```



DEPARTMENT OF COMPUTERSCIENCE & ENGINEERING

Discover. Learn. Empower.

```
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];
```

```
    }
```

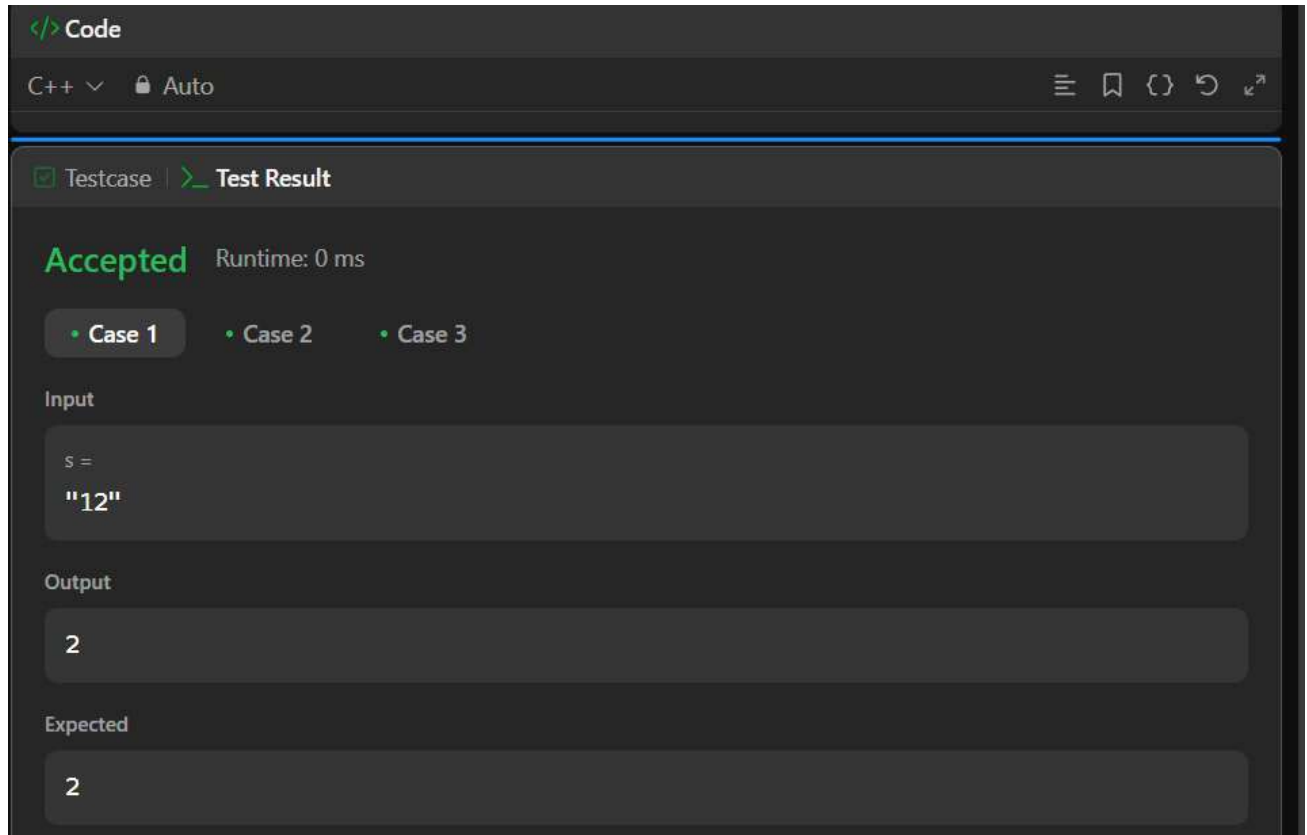
```
}
```

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

```
}
```

```
};
```

4.Output:



The screenshot shows a C++ IDE interface. At the top, there's a 'Code' tab with a C++ dropdown and an 'Auto' button. Below this, there's a 'Testcase' tab and a 'Test Result' tab. The 'Test Result' tab is active, showing 'Accepted' in green text and 'Runtime: 0 ms'. Below this, there are three tabs: 'Case 1', 'Case 2', and 'Case 3'. The 'Case 1' tab is selected. Under 'Case 1', there are three sections: 'Input', 'Output', and 'Expected'. The 'Input' section shows 'S =' followed by a text box containing '"12"'. The 'Output' section shows a text box containing '2'. The 'Expected' section shows a text box containing '2'.

Fig.7:Decode Ways

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:

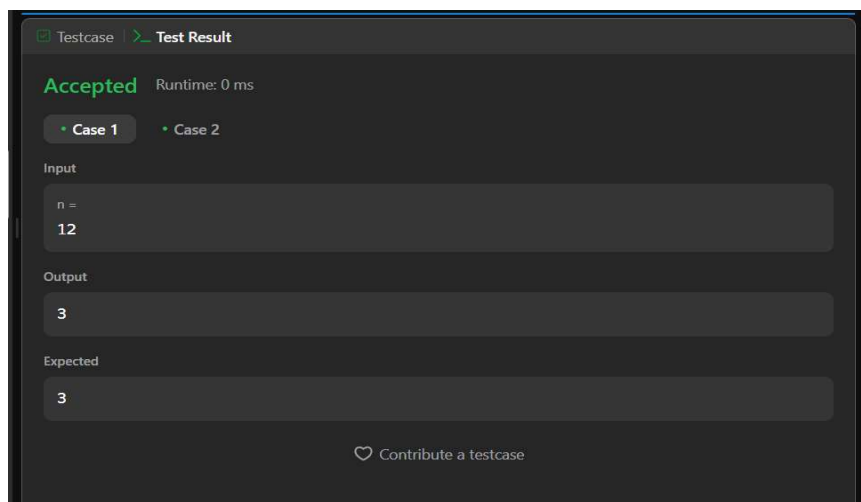


Fig.8: Perfect Squares



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:

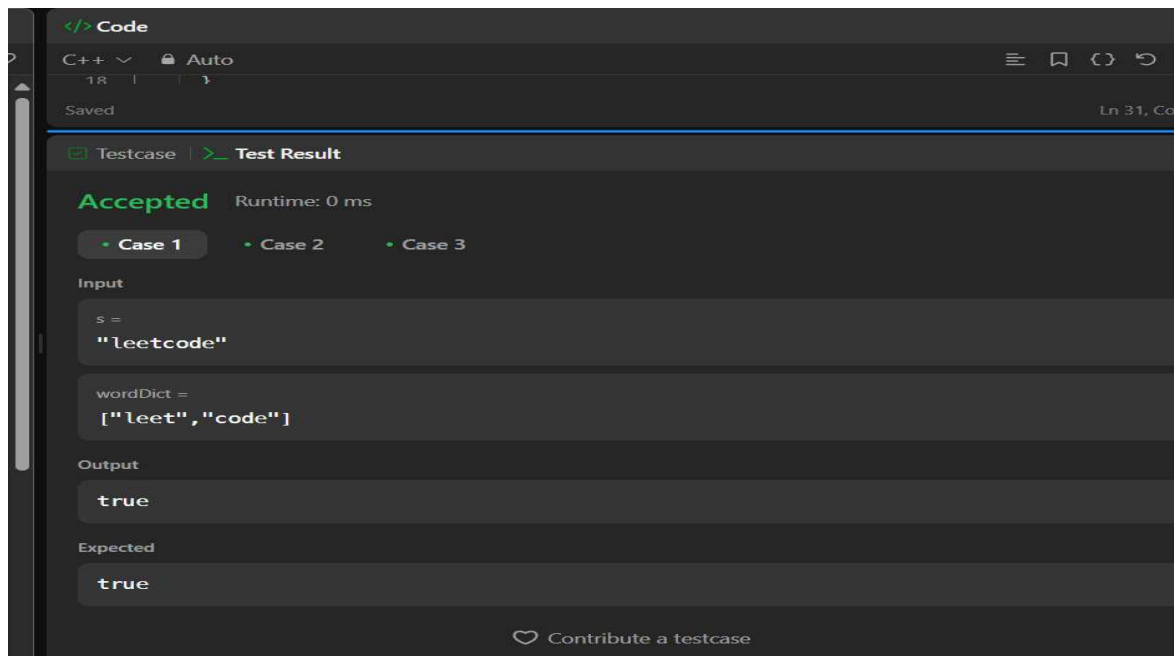


Fig.9: Word Break