Experiment - 7

Name: Aryan Anand
UID: 22BET10056
Branch: BE-IT
Section: 22BET_IOT_702/B
Semester: 6th
DOP: 20/03/25
Subject: AP LAB-2
Subject Code: 22ITH-351

1. House Robber

- **Aim:** To maximize the amount of money robbed from non-adjacent houses.

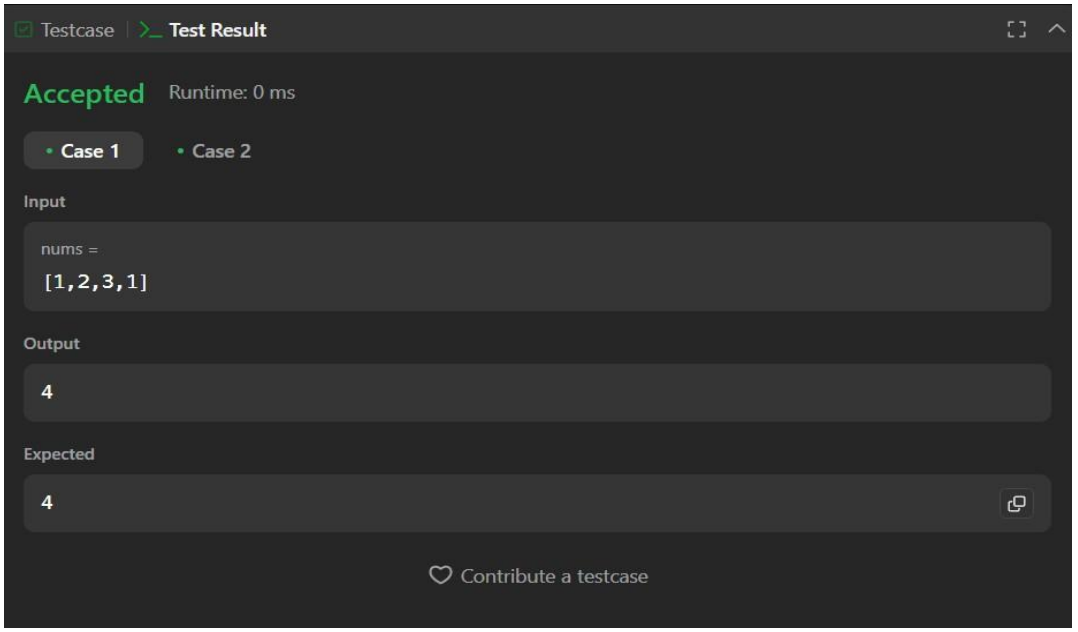
- **Objective:**

1. Use dynamic programming with dp[i] = max(dp[i-1], dp[i-2] + nums[i]).
2. Optimize space using two variables.

- **Code (C++):**

```
#include <bits/stdc++.h> using
namespace std;
 //ARYAN ANAND
int rob(vector<int>& nums) {
int prev1 = 0, prev2 = 0;     for
(int num : nums) {
       int curr = max(prev1, prev2 + num);
       prev2 = prev1;
prev1 = curr;
   }
   return prev1;
}

int main() {     vector<int> nums =
{2,7,9,3,1};     cout << rob(nums)
<< endl;
   return 0;
}
```

Output :



2. Jump Game

- **Aim:** To determine if you can reach the last index of an array using jump values.

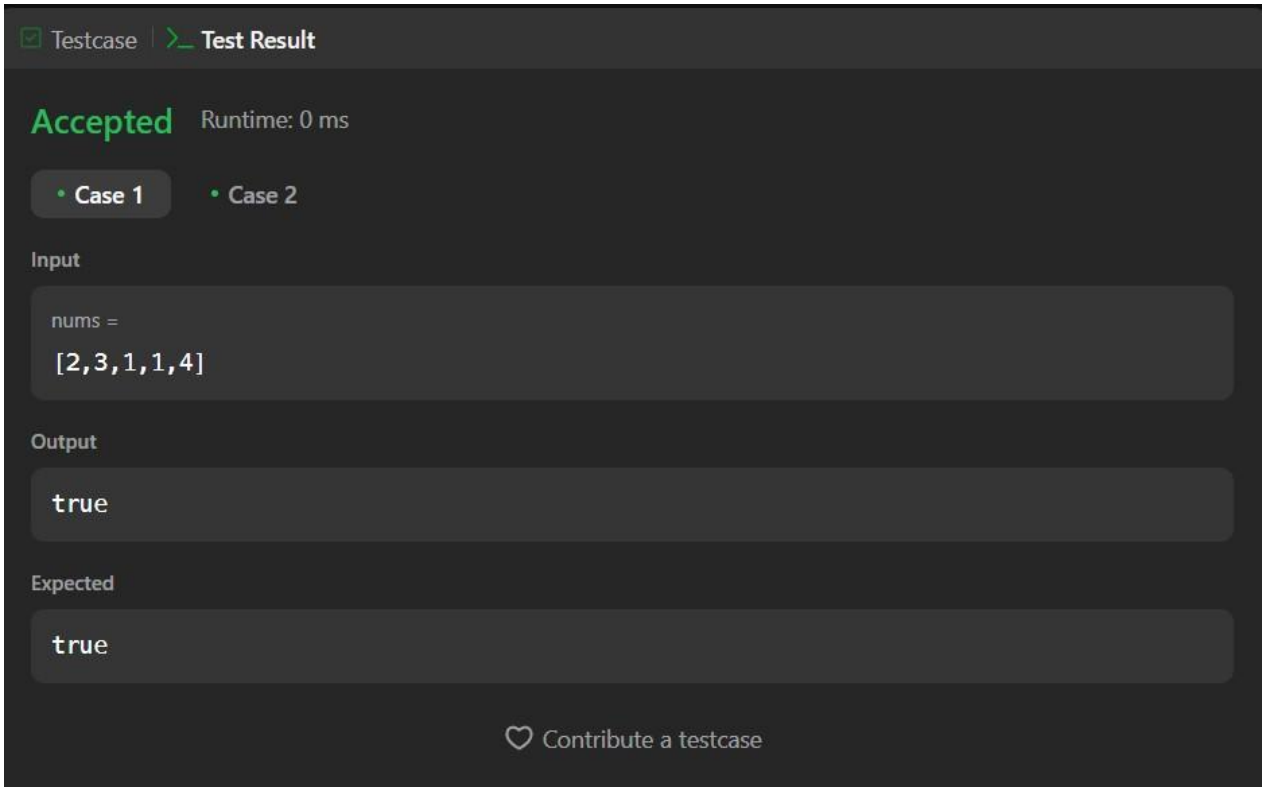
- **Objective:**

1. Maintain maxReach and check if the current index is reachable.
2. Update maxReach = max(maxReach, i + nums[i]).

- **Code (C++):**

```
#include <bits/stdc++.h> using
namespace std;

bool canJump(vector<int>& nums) {
int maxReach = 0;     for (int i = 0; i <
nums.size(); i++) {         if (i >
maxReach) return false;
        maxReach = max(maxReach, i + nums[i]);
    }
    return true;
}
//ARYAN ANAND
int main() {
    vector<int> nums = {2,3,1,1,4};
    cout << (canJump(nums) ? "true" : "false") << endl;
return 0;
}
```

Output :



3. Unique Paths

- **Aim:** To count the number of unique paths to reach the bottom-right corner of a grid.

- **Objective:**

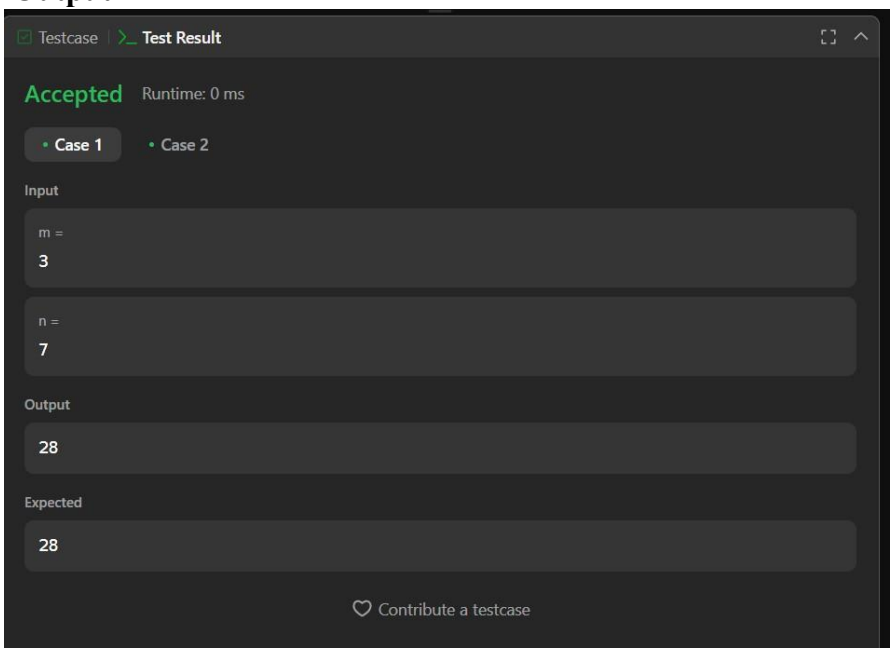
1. Use DP with dp[i][j] = dp[i-1][j] + dp[i][j-1].
2. Optimize space using a 1D array.

- **Code (C++):**

```
#include <bits/stdc++.h>
using namespace std;

int uniquePaths(int m, int n) {
vector<int> dp(n, 1);     for
(int i = 1; i < m; i++) {
for (int j = 1; j < n; j++) {
dp[j] += dp[j - 1];
      }
}
   return dp[n - 1];
}

int main() {     int
m = 3, n = 7;
```

```
    cout << uniquePaths(m, n) << endl;
return 0;
}
```

Output :



4. Coin Change

- **Aim:** To find the minimum number of coins required to make up a given amount.

- **Objective:**

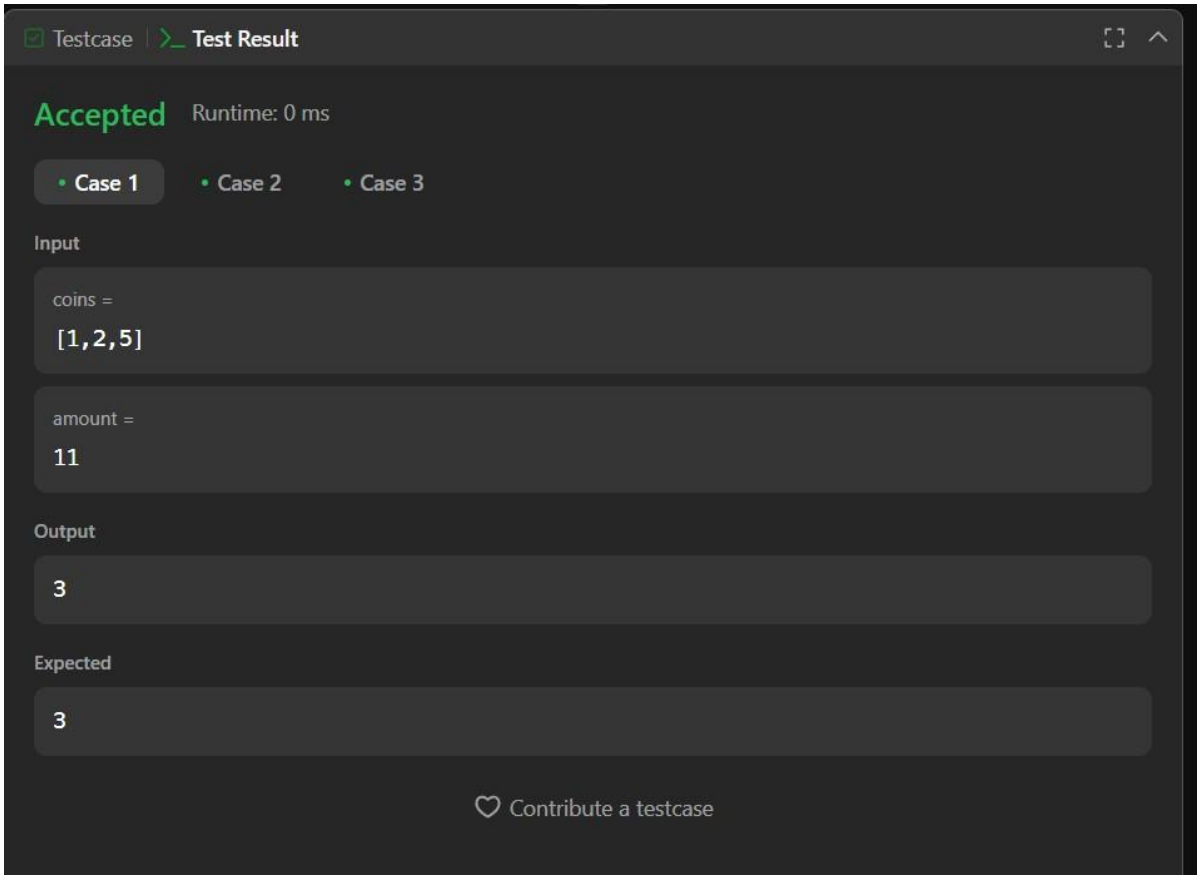
1. Use bottom-up DP with dp[i] = min(dp[i], 1 + dp[i - coin]).
2. If dp[amount] == INF, return -1.

- **Code (C++):**

```
#include <bits/stdc++.h>
using namespace std;

int coinChange(vector<int>& coins, int amount) {
    vector<int> dp(amount + 1, INT_MAX);
dp[0] = 0;     for (int coin : coins) {        for
(int i = coin; i <= amount; i++) {           if
(dp[i - coin] != INT_MAX)                dp[i]
= min(dp[i], 1 + dp[i - coin]);
        }
    }
    return dp[amount] == INT_MAX ? -1 : dp[amount]; }

int main() {
```

```
    vector<int> coins = {1,2,5};
int amount = 11;
    cout << coinChange(coins, amount) << endl;
return 0;
}
```

Output :



5. Longest Increasing Subsequence

- **Aim:** To find the length of the longest increasing subsequence in an array.

- **Objective:**

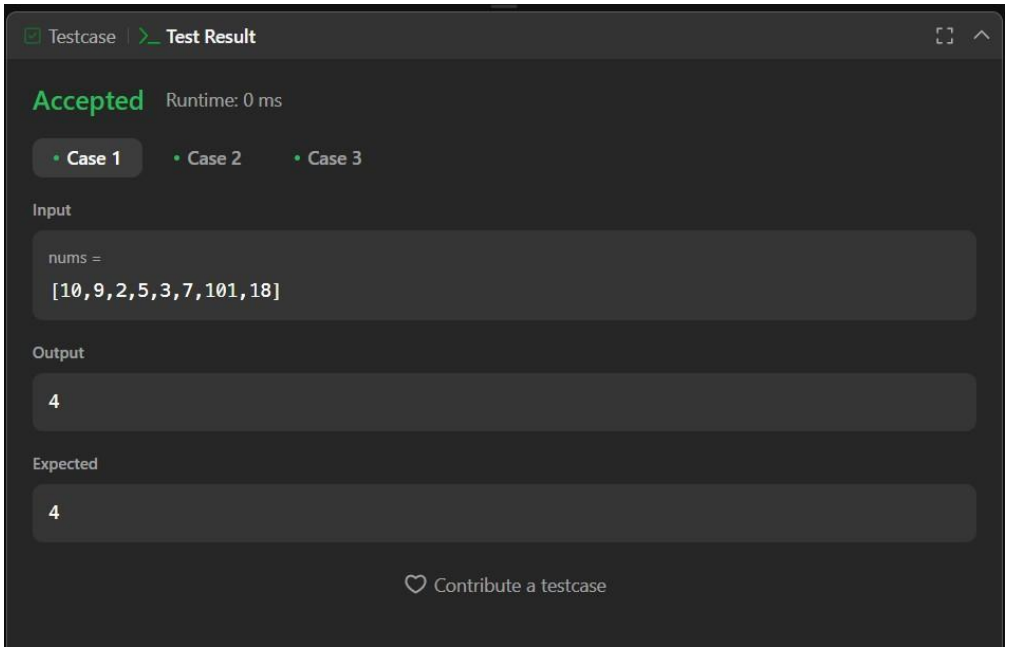
1. Use dynamic programming with dp[i] = max(dp[i], dp[j] + 1) if nums[i] > nums[j].
2. Optimize using binary search and lower_bound.

- **Code (C++)**

```
#include <bits/stdc++.h>
using namespace std;
  //ARYAN ANAND
int lengthOfLIS(vector<int>& nums) {
    vector<int> sub;
for (int num : nums) {
        auto it = lower_bound(sub.begin(), sub.end(), num);
if (it == sub.end()) sub.push_back(num);
```

```
        else *it = num;
    }
    return sub.size();
}

int main() {
    vector<int> nums = {10,9,2,5,3,7,101,18};
    cout << lengthOfLIS(nums) << endl;
return 0;
}
```

Output :



6. Maximum Product Subarray

- **Aim:** To find the contiguous subarray with the maximum product.

- **Objective:**

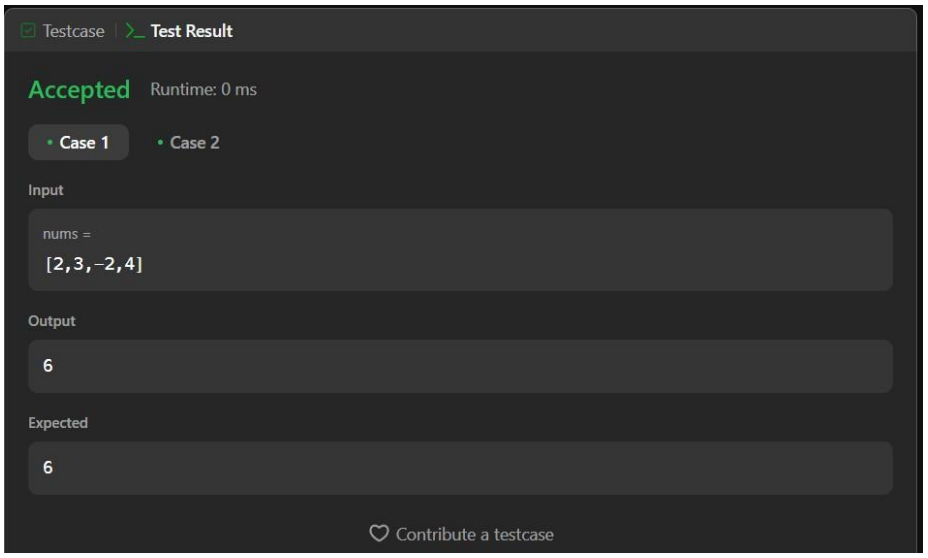
1. Maintain maxProd and minProd to track positive and negative numbers.
2. Update maxProd = max(nums[i], nums[i] * minProd, nums[i] * maxProd).

• **Code (C++):**

```
#include <bits/stdc++.h>
using namespace std;

int maxProduct(vector<int>& nums) {
    int maxProd = nums[0], minProd = nums[0], result = nums[0];
    for (int i = 1; i < nums.size(); i++) {        if
(nums[i] < 0) swap(maxProd, minProd);
maxProd = max(nums[i], maxProd * nums[i]);
minProd = min(nums[i], minProd * nums[i]);
```

```
        result = max(result, maxProd);
    }
    return result;
}

int main() {     vector<int> nums =
{2,3,-2,4};     cout <<
maxProduct(nums) << endl;     return
0;
}
```

Output :



7. Decode Ways

- **Aim:** To count the number of ways to decode a string of digits into letters.

- **Objective:**

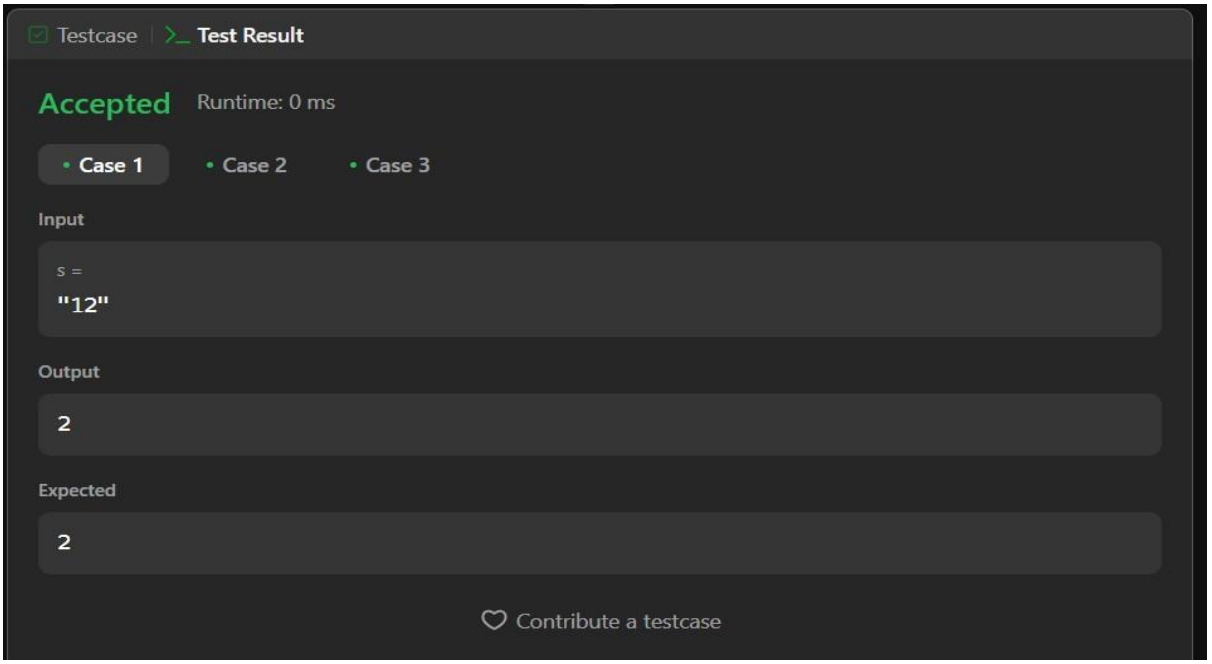
1. Use dynamic programming with dp[i] = dp[i-1] + dp[i-2] based on valid decodings.
2. Handle edge cases for '0'.

- **Code (C++):**

```
#include <bits/stdc++.h>
using namespace std;

int numDecodings(string s) {     if
(s.empty() || s[0] == '0') return 0;
int n = s.size();     vector<int> dp(n +
1, 0);     dp[0] = dp[1] = 1;

    for (int i = 2; i <= n; i++) {         if (s[i
- 1] != '0') dp[i] += dp[i - 1];         int
twoDigit = stoi(s.substr(i - 2, 2));
        if (twoDigit >= 10 && twoDigit <= 26) dp[i] += dp[i - 2];
```

```
    }
    return dp[n];
}

int main() {
string s = "226";
    cout << numDecodings(s) << endl;
return 0;
}
```

Output :



8. Best Time to Buy and Sell a Stock with Cooldown

- **Aim:** To maximize profit with a cooldown after selling a stock.

- **Objective:**

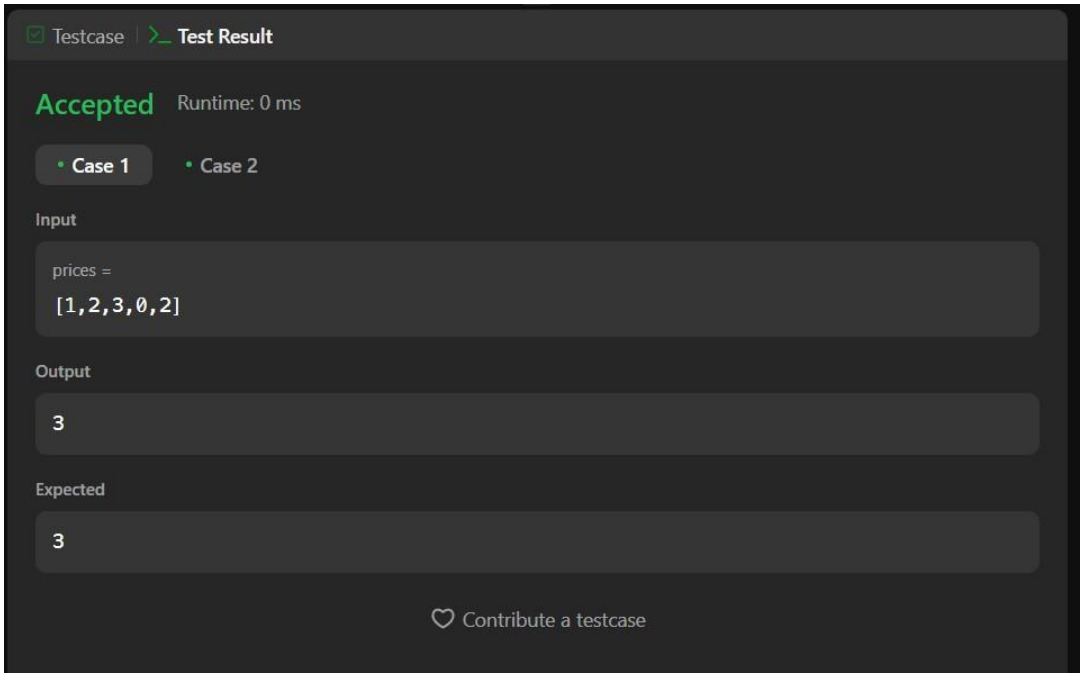
1. Use DP states: buy[i], sell[i], cooldown[i].
2. Transition:
3. buy[i] = max(buy[i-1], cooldown[i-1] - price)
4. sell[i] = max(sell[i-1], buy[i-1] + price)

• Code (C++):

```
#include <bits/stdc++.h>
using namespace std;

int maxProfit(vector<int>& prices) {     int
buy = INT_MIN, sell = 0, cooldown = 0;
    for (int price : prices) {         int
prevSell = sell;         sell = max(sell,
buy + price);         buy = max(buy,
```

```
cooldown - price);        cooldown =
prevSell;
   }     return
sell;
}

int main() {     vector<int> prices =
{1,2,3,0,2};     cout <<
maxProfit(prices) << endl;
   return 0;
}
```

Output :



9. Perfect Squares

- **Aim:** To find the minimum number of perfect squares summing up to n.

- **Objective:**

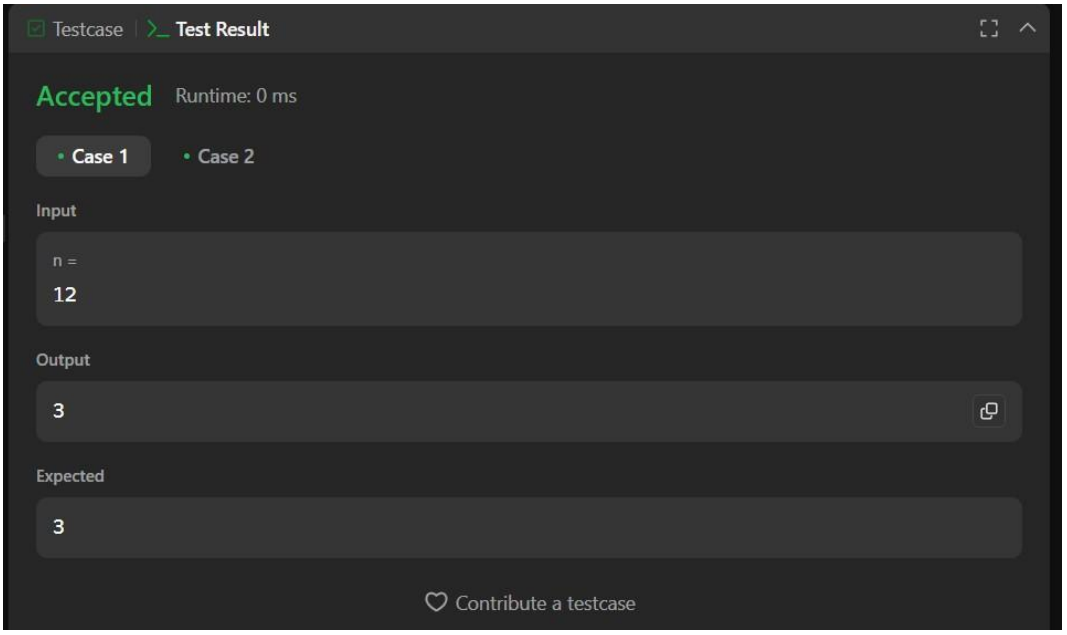
1. Use dynamic programming with dp[i] = min(dp[i], 1 + dp[i - square]).
2. Iterate over all perfect squares up to n.

- **Code (C++)**

```
#include <bits/stdc++.h>
using namespace std;

int numSquares(int n) {
vector<int> dp(n + 1, INT_MAX);
dp[0] = 0;
```

```
    for (int i = 1; i * i <= n; i++) {
int square = i * i;         for (int j =
square; j <= n; j++) {
            dp[j] = min(dp[j], 1 + dp[j - square]);
        }
}
    return dp[n];
}

int main() {
    int n = 12;
    cout << numSquares(n) << endl;
return 0;
}
```

Output :



10. Word Break

- **Aim:** To determine if a string can be segmented into a space-separated sequence of dictionary words.

- **Objective:**

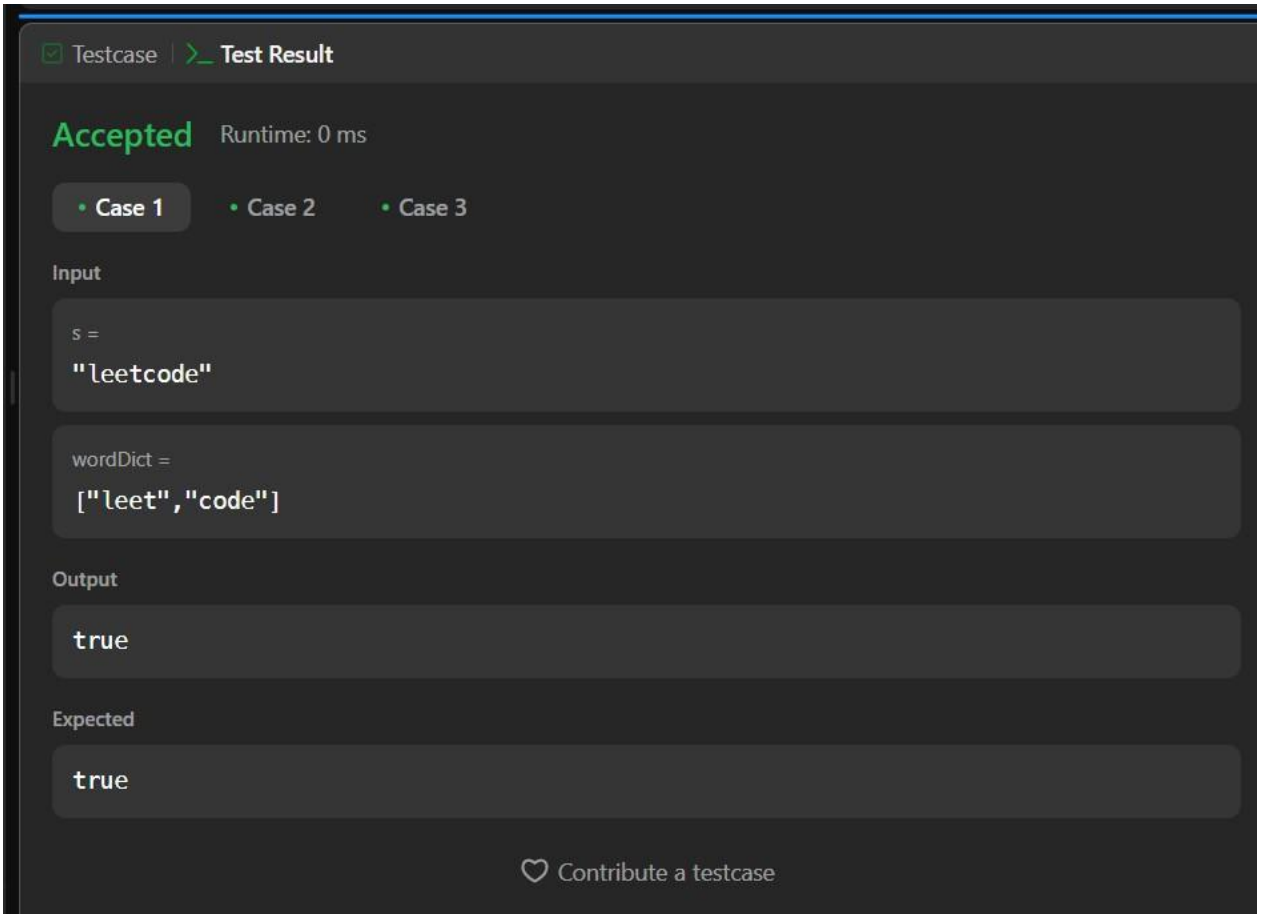
1. Use dynamic programming with dp[i] = true if dp[j] && wordDict contains substring(j, i).
2. Iterate j over possible partitions.

- **Code (C++):**

```
#include <bits/stdc++.h>
using namespace std;

bool wordBreak(string s, vector<string>& wordDict) {
unordered_set<string> dict(wordDict.begin(), wordDict.end());
vector<bool> dp(s.size() + 1, false);     dp[0] = true;
```

```
    for (int i = 1; i <= s.size(); i++) {
for (int j = 0; j < i; j++) {
            if (dp[j] && dict.find(s.substr(j, i - j)) != dict.end()) {
dp[i] = true;                 break;
            }
        }
}
    return dp[s.size()];
}

int main() {
    string s = "leetcode";
    vector<string> wordDict = {"leet", "code"};
    cout << (wordBreak(s, wordDict) ? "true" : "false") << endl;
return 0;
}
```

Output :



11. Word Break II

- **Aim:** To return all possible sentences that can be formed from a given string using a dictionary.
- **Objective:**

1. Use backtracking to explore all partitions of the string.

2. Store computed results using memoization.

• **Code (C++):**

```
#include <bits/stdc++.h>
using namespace std;

unordered_map<string, vector<string>> memo;
  //ARYAN ANAND
vector<string> wordBreakHelper(string s, unordered_set<string>& dict) {
if (memo.count(s)) return memo[s];
    if (s.empty()) return {""};

    vector<string> result;     for (int i
= 1; i <= s.size(); i++) {         string
word = s.substr(0, i);         if
(dict.count(word)) {
            vector<string> suffixes = wordBreakHelper(s.substr(i), dict);
for (string suffix : suffixes) {
                result.push_back(word + (suffix.empty() ? "" : " ") + suffix);
            }
        }
    }
    return memo[s] = result;
}

vector<string> wordBreak(string s, vector<string>& wordDict) {
unordered_set<string> dict(wordDict.begin(), wordDict.end());     return
wordBreakHelper(s, dict);
}

int main() {     string s =
"catsanddog";
    vector<string> wordDict = {"cat", "cats", "and", "sand", "dog"};
vector<string> result = wordBreak(s, wordDict);     for (string
sentence : result) cout << sentence << endl;     return 0;
}
```

Output :


Testcase
Test Result
Accepted
Runtime: 0 ms
Case 1
Case 2
Case 3
Input
s =
"catsanddog"
wordDict =
["cat","cats","and","sand","dog"]
Output
["cat sand dog","cats and dog"]
Expected
["cats and dog","cat sand dog"]
Contribute a testcase