Experiment 6

Student Name: Aman Bansal **UID: 22BCS13365**
Branch: UIE CSE 3rd Year **Section/Group: 22BCS_KPIT-901-'B'**
Semester: 6th **Date of Performance: 16th Mar 2025**
Subject Name: Advanced Programming – II **Subject Code: 22CSP-351**

1. Aim:

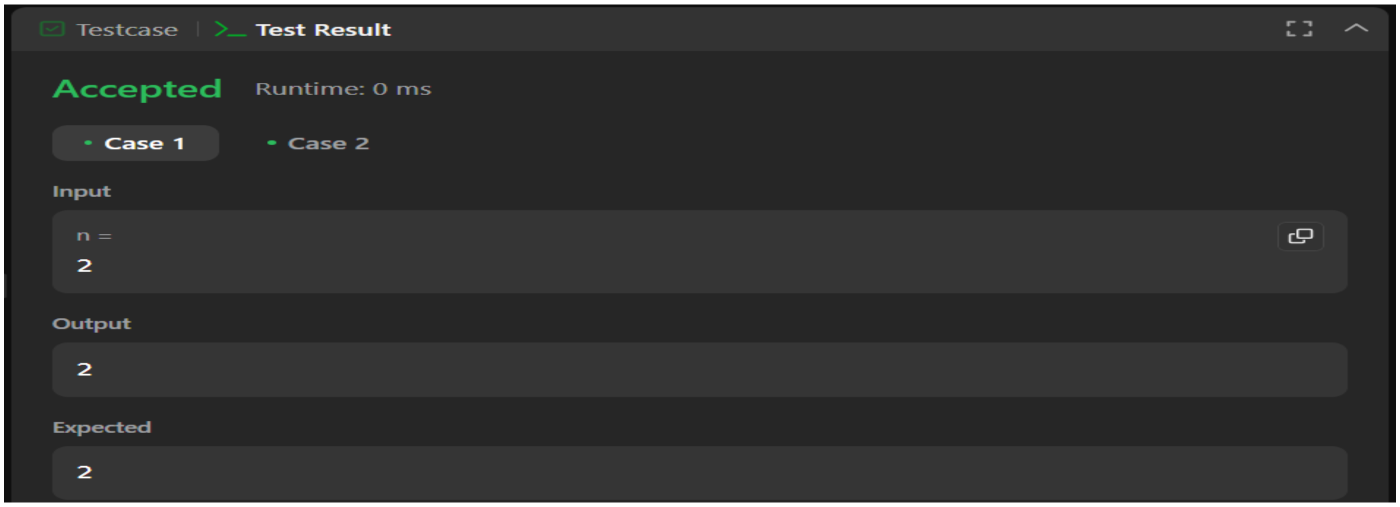
You are climbing a staircase. It takes n steps to reach the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

2. Implementation/Code:

```
class Solution {

public:

    int climbStairs(int n) {

        vector<int>dp(n+1);

        dp[0]=1,dp[1]=1;

        for(int i=2;i<=n;i++){

            dp[i]=dp[i-1]+dp[i-2];

        }

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

3. **Output:**



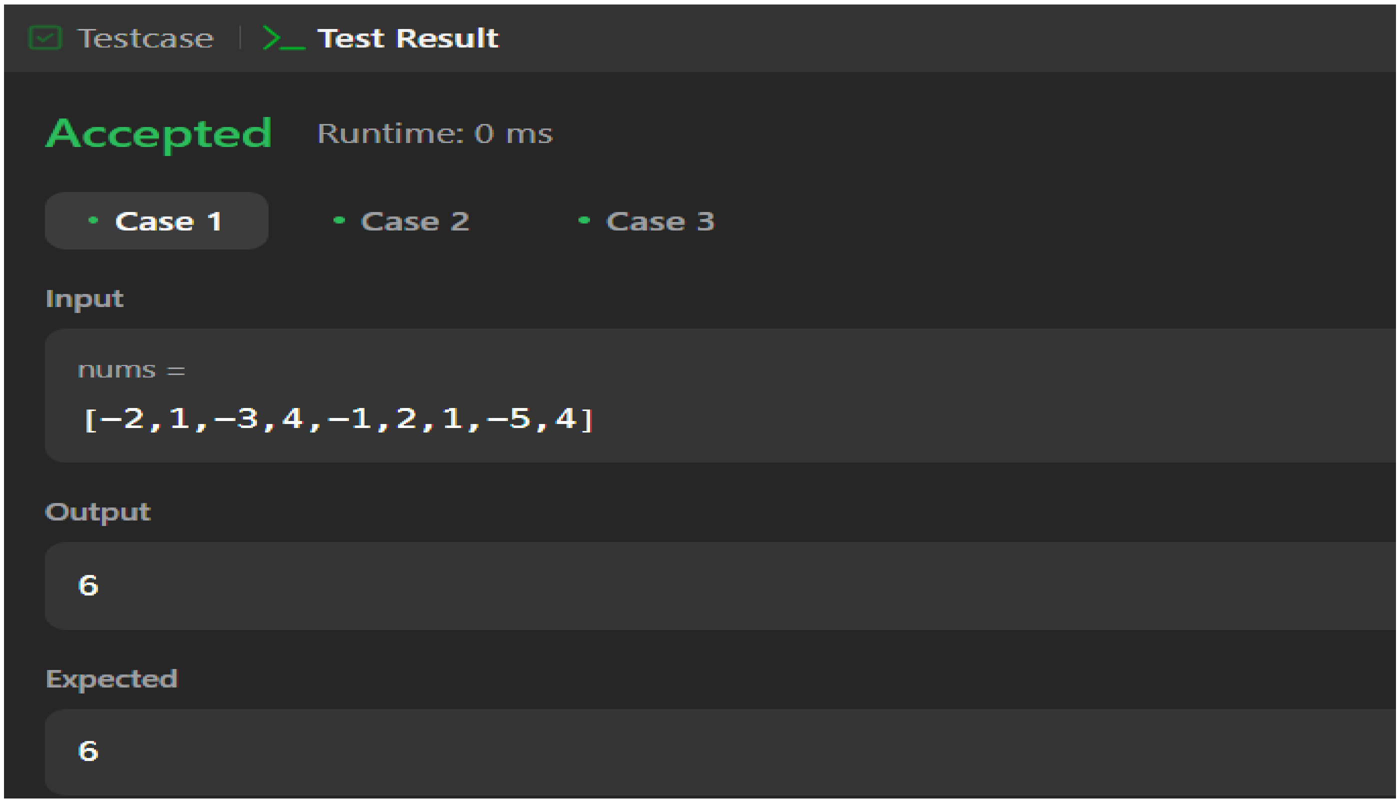
```
Accepted    Runtime: 0 ms
• Case 1    • Case 2
Input
n =
2
Output
2
Expected
2
```

QUES:2

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

```
class Solution {

public:

    int maxSubArray(vector<int>& nums) {

        int sum=0;

        int maxi=nums[0];

        for(int i=0;i<nums.size();i++){

            sum+=nums[i];
            maxi=max(maxi,sum);
            if(sum<0)sum=0;
        }
        return maxi;
    }
};
```

3. Output:


Testcase | >_ Test Result

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

```
nums =
[-2,1,-3,4,-1,2,1,-5,4]
```

Output

```
6
```

Expected

```
6
```

QUESTION:3 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.

CODE:

```cpp
class Solution {
public:
    bool canJump(vector<int>& nums) {
        int index=0;
        int targetIndex=nums.size()-1;
        for(int i=0;i<nums.size();i++){
            if(i>index)return false;
            index=max(index,i+nums[i]);
            if(index>=targetIndex)return true;
        }
        return false;
    }
};
```

QUESTION:4 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.

CODE:

```
class Solution {
public:
    int rob(vector<int>& nums) {
        int n=nums.size();
        if(n==1) return nums[0];
        if(n==2) return (nums[1]>nums[0])?nums[1]:nums[0];
        int m_money=0;
        vector<int>max_money(n);
        max_money[0]=nums[0];
        max_money[1]=(nums[1]>nums[0])?nums[1]:nums[0];
        for(int i=2;i<nums.size();i++){
            max_money[i]=max(max_money[i-1],nums[i]+max_money[i-2]);
        }
        return max_money[n-1];
    }
};
```

QUESTION:5 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*.

CODE:

```
class Solution {
public:
    int uniquePaths(int m, int n) {
        vector<int> cur(n, 1);
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                cur[j] += cur[j - 1];
            }
        }
        return cur[n - 1];
    }
};
```

QUESTION:6 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.

CODE:

```
class Solution {
    public int coinChange(int[] coins, int amount) {
        int[] minCoins = new int[amount + 1];
        Arrays.fill(minCoins, amount + 1);
        minCoins[0] = 0;

        for (int i = 1; i <= amount; i++) {
            for (int j = 0; j < coins.length; j++) {
                if (i - coins[j] >= 0) {
                    minCoins[i] = Math.min(minCoins[i], 1 + minCoins[i - 
coins[j]]);
                }
            }
        }

        return minCoins[amount] != amount + 1 ? minCoins[amount] : -1;
    }
}
```

QUESTION:7 Given an integer array nums, return *the length of the longest strictly increasing subsequence*.

CODE:

```
class Solution {
public:
    int lengthOfLIS(vector<int>& nums) {
        vector<int> res;

        for (int n : nums) {
            if (res.empty() || res.back() < n) {
                res.push_back(n);
            } else {
                int idx = binarySearch(res, n);
                res[idx] = n;
            }
        }

        return res.size();
    }

private:
    int binarySearch(const vector<int>& arr, int target) {
        int left = 0;
        int right = arr.size() - 1;

        while (left <= right) {
            int mid = (left + right) / 2;
            if (arr[mid] == target) {
                return mid;
            } else if (arr[mid] > target) {
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }

        return left;
    }
};
```

QUESTION:8 Given an integer array nums, find a subarray that has the largest product, and return *the product*.

The test cases are generated so that the answer will fit in a 32-bit integer

CODE:

```
class Solution {
public:
    int maxProduct(vector<int>& nums) {
        int maxi = INT_MIN;
        int prod=1;
        for(int i=0;i<nums.size();i++)
        {
          prod*=nums[i];
          maxi=max(prod,maxi);
          if(prod==0)
           prod=1;
        }
        prod=1;
        for(int i=nums.size()-1;i>=0;i--)
        {
          prod*=nums[i];

          maxi=max(prod,maxi);
          if(prod==0)
           prod=1;
        }
        return maxi;
    }
};
```

QUESTION:9 You have intercepted a secret message encoded as a string of numbers. The message is decoded via the following mapping:

CODE:

```
class Solution {
public:
    int numDecodings(string s) {
        if (s[0] == '0') {
            return 0;
        }

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

        for (int i = 2; i <= n; i++) {
            int one = s[i - 1] - '0';
            int two = stoi(s.substr(i - 2, 2));
            if (1 <= one && one <= 9) {
                dp[i] += dp[i - 1];
            }
            if (10 <= two && two <= 26) {
                dp[i] += dp[i - 2];
            }
        }

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

QUESTION 10: 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.

CODE:

```
class Solution
{
public:
    int numSquares(int n)
    {
        if (n <= 0)
            return 0;
        vector<int> cntPerfectSquares(n + 1, INT_MAX);
        cntPerfectSquares[0] = 0;
        for (int i = 1; i <= n; i++)
        {
            for (int j = 1; j*j <= i; j++)
            {
                cntPerfectSquares[i] =
                min(cntPerfectSquares[i], cntPerfectSquares[i - j*j] + 1);
            }
        }
        return cntPerfectSquares.back();
    }
};
```

QUESTION 11: 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.

CODE:

```
class Solution {
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        vector<bool> dp(s.size() + 1, false);
        dp[0] = true;

        for (int i = 1; i <= s.size(); i++) {
            for (const string& w : wordDict) {
                int start = i - w.length();
                if (start >= 0 && dp[start] && s.substr(start, w.length()) == w) {
                    dp[i] = true;
                    break;
                }
            }
        }
        return dp[s.size()];
    }
};
```