DOMAIN WINTER CAMP

DAY - 8

Student Name: Navneet **UID:** 22BCS15680

Branch: BE-CSE **Section:** 22BCS_FL_IOT-603/B

Dynamic Programming

Very Easy:

1. N-th Tribonacci Number

The Tribonacci sequence Tn is defined as follows:
T0 = 0, T1 = 1, T2 = 1, and Tn+3 = Tn + Tn+1 + Tn+2 for n >= 0.
Given n, return the value of Tn.

Solution:

```
#include <iostream>
using namespace std;

class Solution {
public:
    int tribonacci(int n) {
        if (n == 0) return 0;
        if (n == 1 || n == 2) return 1;

        int t0 = 0, t1 = 1, t2 = 1;
        int t3;
        for (int i = 3; i <= n; ++i) {
            t3 = t0 + t1 + t2;
            t0 = t1;
            t1 = t2;
            t2 = t3;
        }
        return t3;
    }
};

int main() {
    Solution solution;
    int n;
    cout << "Enter number: ";
    cin >> n;
    cout << "Tribonacci number: " << solution.tribonacci(n) << endl;
    return 0;
}
```

Output:

```
Enter number: 25
Tribonacci number: 1389537


=== Code Execution Successful ===
```

Easy:

2. Climbing Stairs

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?

Solution:

```
#include <iostream>
using namespace std;

class Solution {
public:
    int climbStairs(int n) {
        if (n == 1) return 1;

        int prev1 = 1, prev2 = 2;
        for (int i = 3; i <= n; ++i) {
            int curr = prev1 + prev2;
            prev1 = prev2;
            prev2 = curr;
        }
        return prev2;
    }
};

int main() {
    Solution solution;
    int n;
    cout << "Enter number of steps: ";
    cin >> n;
    cout << "Distinct ways to climb to the top: " << solution.climbStairs(n) << endl;
    return 0;
}
```

Output:

```
Enter number of steps: 3
Distinct ways to climb to the top: 3


=== Code Execution Successful ===
```

Medium:

3. Longest Palindromic Substring

Given a string s, return the longest palindromic substring in s.

Solution:

```
#include <iostream>
#include <string>
using namespace std;

class Solution {
public:
    string longestPalindrome(string s) {
        int start = 0, maxLength = 0;
        for (int i = 0; i < s.size(); ++i) {
            expandAroundCenter(s, i, i, start, maxLength);
            expandAroundCenter(s, i, i + 1, start, maxLength);
        }
        return s.substr(start, maxLength);
    }

private:
    void expandAroundCenter(const string& s, int left, int right, int& start, int&
maxLength) {
        while (left >= 0 && right < s.size() && s[left] == s[right]) {
            int length = right - left + 1;
            if (length > maxLength) {
                maxLength = length;
                start = left;
            }
            --left;
            ++right;
        }
    }
};

int main() {
    Solution solution;
    string s;
    cout << "Enter a string: ";
    cin >> s;
    cout << "Longest palindromic substring: " << solution.longestPalindrome(s) <<
endl;
    return 0;
}
```

Output:

```
Enter a string: babad
Longest palindromic substring: bab


=== Code Execution Successful ===
```

Hard:

4. Maximal Rectangle

Given a rows x cols binary matrix filled with 0's and 1's, find the largest rectangle containing only 1's and return its area.

Solution:

```
#include <iostream>
#include <vector>
#include <stack>
#include <string>
using namespace std;

class Solution {
public:
    int maximalRectangle(vector<vector<char>>& matrix) {
        if (matrix.empty() || matrix[0].empty()) return 0;

        int rows = matrix.size();
        int cols = matrix[0].size();
        vector<int> heights(cols, 0);
        int maxArea = 0;

        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                heights[j] = matrix[i][j] == '1' ? heights[j] + 1 : 0;
            }
            maxArea = max(maxArea, largestRectangleArea(heights));
        }

        return maxArea;
    }

    int largestRectangleArea(vector<int>& heights) {
        stack<int> s;
        int maxArea = 0;
        heights.push_back(0);

        for (int i = 0; i < heights.size(); i++) {
            while (!s.empty() && heights[i] < heights[s.top()]) {
                int h = heights[s.top()];
                s.pop();
                int width = s.empty() ? i : i - s.top() - 1;
                maxArea = max(maxArea, h * width);
```

```
            }
            s.push(i);
        }

        return maxArea;
    }
};

int main() {
    vector<vector<char>> matrix = {
        {'1', '0', '1', '0', '0'},
        {'1', '0', '1', '1', '1'},
        {'1', '1', '1', '1', '1'},
        {'1', '0', '0', '1', '0'}
    };

    Solution solution;
    int result = solution.maximalRectangle(matrix);
    cout << "Maximal rectangle area: " << result << endl;

    return 0;
}
```

Output:

```
Maximal rectangle area: 6


=== Code Execution Successful ===
```

Very Hard:

5. Cherry Pickup

You are given an n x n grid representing a field of cherries, each cell is one of three possible integers.
0 means the cell is empty, so you can pass through,
1 means the cell contains a cherry that you can pick up and pass through, or
-1 means the cell contains a thorn that blocks your way.
Return the maximum number of cherries you can collect by following the rules below:
Starting at the position (0, 0) and reaching (n - 1, n - 1) by moving right or down through valid path cells (cells with value 0 or 1).
After reaching (n - 1, n - 1), returning to (0, 0) by moving left or up through valid path cells.
When passing through a path cell containing a cherry, you pick it up, and the cell becomes an empty cell 0.
If there is no valid path between (0, 0) and (n - 1, n - 1), then no cherries can be collected.

Solution:

```
#include <iostream>
```

```
#include <vector>
#include <algorithm>
using namespace std;

class Solution {
public:
    int cherryPickup(vector<vector<int>>& grid) {
        int n = grid.size();

        // DP table: dp[step][r1][r2] = max cherries collected by two people at step 'step'
        vector<vector<vector<int>>> dp(2 * n - 1, vector<vector<int>>(n, vector<int>(n,
-1)));

        // Initialize the base case
        dp[0][0][0] = grid[0][0]; // Starting position for both people

        // Fill the DP table for each step
        for (int step = 1; step < 2 * n - 1; ++step) {
            for (int r1 = max(0, step - (n - 1)); r1 <= min(n - 1, step); ++r1) {
                for (int r2 = max(0, step - (n - 1)); r2 <= min(n - 1, step); ++r2) {
                    int c1 = step - r1;
                    int c2 = step - r2;
                    if (grid[r1][c1] == -1 || grid[r2][c2] == -1) continue; // Skip blocked cells

                    int cherries = grid[r1][c1] + (r1 != r2 ? grid[r2][c2] : 0); // Collect
cherries

                    // Explore all possible ways to move from the previous step
                    int maxPrev = -1;
                    for (int prevR1 = r1 - 1; prevR1 <= r1; ++prevR1) {
                        for (int prevR2 = r2 - 1; prevR2 <= r2; ++prevR2) {
                            int prevC1 = step - prevR1;
                            int prevC2 = step - prevR2;
                            if (prevR1 >= 0 && prevR1 < n && prevR2 >= 0 && prevR2 < n
&&
                                prevC1 >= 0 && prevC1 < n && prevC2 >= 0 && prevC2 < n
&&
                                dp[step - 1][prevR1][prevR2] != -1) {
                                maxPrev = max(maxPrev, dp[step - 1][prevR1][prevR2]);
                            }
                        }
                    }

                    if (maxPrev != -1) {
                        dp[step][r1][r2] = max(dp[step][r1][r2], maxPrev + cherries);
                    }
                }
            }
        }

        return dp[2 * n - 2][n - 1][n - 1] == -1 ? 0 : dp[2 * n - 2][n - 1][n - 1];
    }
};
```

```
int main() {
    vector<vector<int>> grid = {
        {0, 1, -1},
        {1, 0, -1},
        {1, 1, 1}
    };

    Solution solution;
    cout << "Maximum cherries picked: " << solution.cherryPickup(grid) << endl;
    return 0;
}
```

Output: