DOMAIN WINTER WINNING CAMP ASSIGNMENT

Student Name: Tamanna Gupta **UID: 22BCS14867**
Branch: BE-CSE::CS201 **Section/Group: 22BCS_FL_IOT-603/B**
Semester: 5th

➢ DAY-8 [27-12-2024]

1. N-th Tribonacci Number *(Very Easy)*

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.

Example 1:

Input: n = 4

Output: 4

Explanation:

T_3 = 0 + 1 + 1 = 2

T_4 = 1 + 1 + 2 = 4

Example 2:

Input: n = 25

Output: 1389537

Constraints:

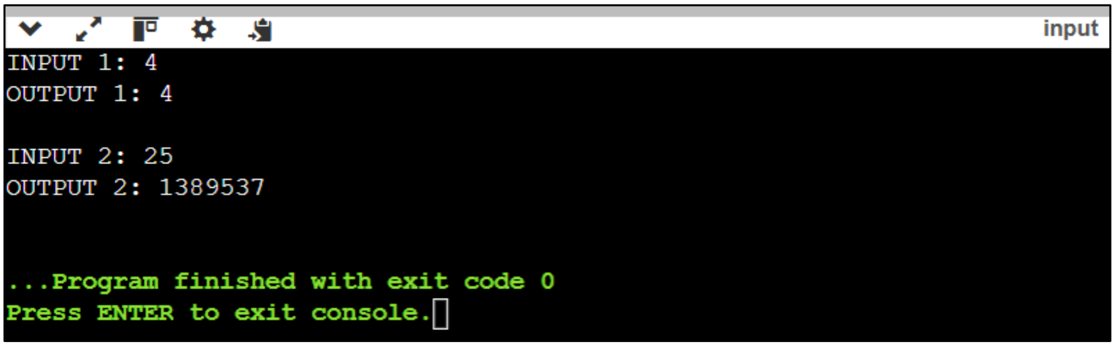
0 <= n <= 37

The answer is guaranteed to fit within a 32-bit integer, ie. answer <= 2^31 - 1.

Implementation/Code:

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

class Solution
{
public:
    int tribonacci(int n)
    {
        if (n == 0) return 0;
        if (n == 1 || n == 2) return 1;
        vector<int> dp(n + 1);
        dp[0] = 0, dp[1] = 1, dp[2] = 1;
        for (int i = 3; i <= n; i++)
        {
            dp[i] = dp[i - 1] + dp[i - 2] + dp[i - 3];
        }
        return dp[n];
    }
};

int main()
{
    Solution sol;
    int n;
    cout<<"INPUT 1: ";
    cin>>n;
    cout <<"OUTPUT 1: "<<sol.tribonacci(n) << endl<<endl;

    cout<<"INPUT 2: ";
    cin>>n;
    cout <<"OUTPUT 2: "<< sol.tribonacci(n) << endl;
    return 0;
}
```

Output:



```
INPUT 1: 4
OUTPUT 1: 4

INPUT 2: 25
OUTPUT 2: 1389537


...Program finished with exit code 0
Press ENTER to exit console.
```

2. Climbing Stairs *(Easy)*

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?

Example 1: **Input: n = 2**
Output: 2
Explanation: There are two ways to climb to the top.
1. 1 step + 1 step
2. 2 steps

Example 2: **Input: n = 3**
Output: 3
Explanation: There are three ways to climb to the top.
1. 1 step + 1 step + 1 step
2. 1 step + 2 steps
3. 2 steps + 1 step

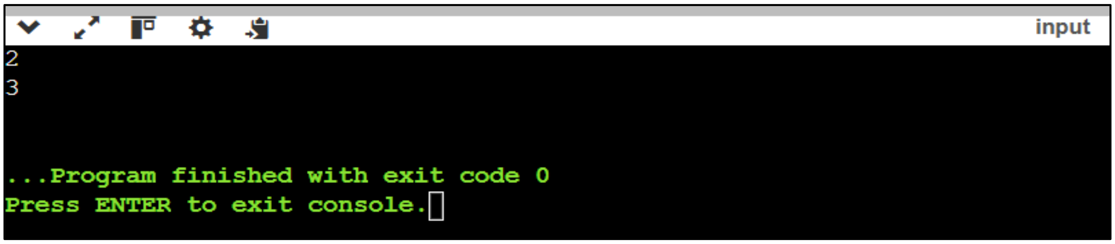
Constraints: 1 <= n <= 45

Implementation/Code:

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

```
class Solution {
public:
    int climbStairs(int n) {
        if (n <= 2) return n;
        vector<int> dp(n + 1);
        dp[1] = 1, dp[2] = 2;
        for (int i = 3; i <= n; i++) {
            dp[i] = dp[i - 1] + dp[i - 2];
        }
        return dp[n];
    }
};

int main() {
    Solution sol;
    cout << sol.climbStairs(2) << endl; // Output: 2
    cout << sol.climbStairs(3) << endl; // Output: 3
    return 0;
}
```

Output:



```
2
3


...Program finished with exit code 0
Press ENTER to exit console.
```

3. Longest Palindromic Substring *(Medium)*

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

<u>Example 1:</u> **Input: s = "babad"**
Output: "bab"
Explanation: "aba" is also a valid answer.

Example 2: Input: s = "cbbd"
Output: "bb"

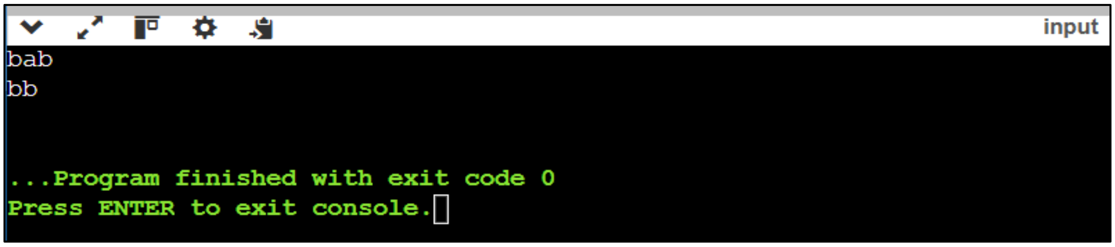
Constraints: 1 <= s.length <= 1000
s consist of only digits and English letters.

Implementation/Code:

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

class Solution {
public:
    string longestPalindrome(string s) {
        int n = s.size();
        if (n == 0) return "";
        vector<vector<bool>> dp(n, vector<bool>(n, false));
        int maxLength = 1, start = 0;

        for (int i = 0; i < n; i++) dp[i][i] = true;

        for (int len = 2; len <= n; len++) {
            for (int i = 0; i <= n - len; i++) {
                int j = i + len - 1;
                if (s[i] == s[j] && (len == 2 || dp[i + 1][j - 1])) {
                    dp[i][j] = true;
                    if (len > maxLength) {
                        maxLength = len;
                        start = i;
                    }
                }
            }
        }
        return s.substr(start, maxLength);
    }
};
```

```
int main() {
    Solution sol;
    cout << sol.longestPalindrome("babad") << endl; // Output: "bab" or "aba"
    cout << sol.longestPalindrome("cbbd") << endl; // Output: "bb"
    return 0;
}
```

Output:

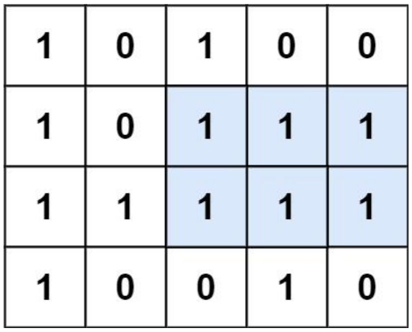


```
bab
bb


...Program finished with exit code 0
Press ENTER to exit console.
```

4. Maximal Rectangle *(Hard)*

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.

Example 1-



1	0	1	0	0
1	0	1	1	1
1	1	1	1	1
1	0	0	1	0

Input:
matrix=
[["1","0","1","0","0"],["1","0","1","1","1"],["1","1","1","1","1"],["1","0","0","1","0"]]
Output: 6
Explanation: The maximal rectangle is shown in the above picture.

Example 2: Input: matrix = [["0"]]
Output: 0

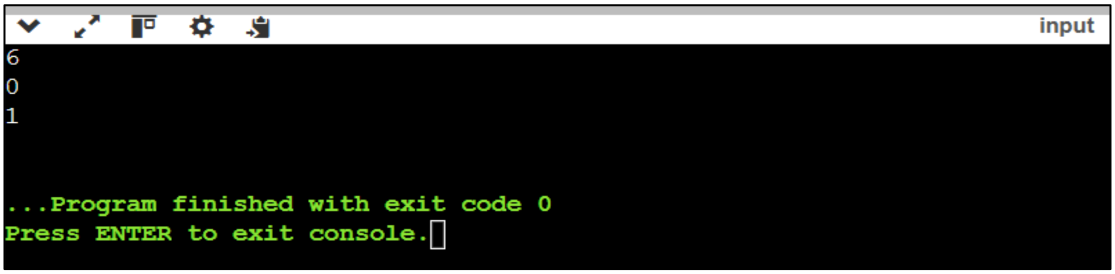
Example 3: Input: matrix = [["1"]]
Output: 1

Constraints: *rows == matrix.length*
cols == matrix[i].length
1 <= row, cols <= 200
matrix[i][j] is '0' or '1'.

Implementation/Code:

```
#include <iostream>
#include <vector>
#include <stack>

using namespace std;

class Solution {
public:
    int maximalRectangle(vector<vector<char>>& matrix) {
        if (matrix.empty()) return 0;
        int m = matrix.size(), n = matrix[0].size();
        vector<int> heights(n, 0);
        int maxArea = 0;

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

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

        for (int i = 0; i < heights.size(); i++) {
            while (!st.empty() && heights[i] < heights[st.top()]) {
                int h = heights[st.top()];
                st.pop();
                int w = st.empty() ? i : i - st.top() - 1;
                maxArea = max(maxArea, h * w);
            }
            st.push(i);
        }
        return maxArea;
    }
};

int main()
{
    Solution sol;
    vector<vector<char>> matrix1 = {
        {'1', '0', '1', '0', '0'},
        {'1', '0', '1', '1', '1'},
        {'1', '1', '1', '1', '1'},
        {'1', '0', '0', '1', '0'}
    };
    cout << sol.maximalRectangle(matrix1) << endl;
    vector<vector<char>> matrix2 = {
        {'0'}
    };
    cout << sol.maximalRectangle(matrix2) << endl;
    vector<vector<char>> matrix3 = {
        {'1'}
    };
    cout << sol.maximalRectangle(matrix3) << endl;
    return 0;
}
```

Output:



```
6
0
1


...Program finished with exit code 0
Press ENTER to exit console.
```

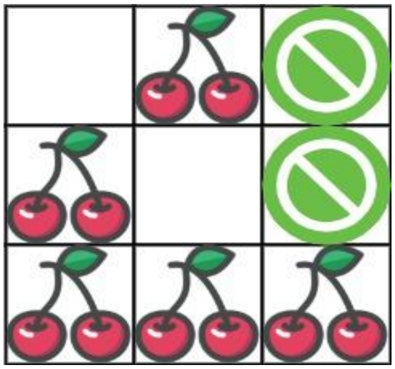
5. Cherry Pickup *(Very Hard)*

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.
- -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.



Example 1:

Input: grid = [[0,1,-1],[1,0,-1],[1,1,1]]
Output: 5
Explanation: The player started at (0, 0), went down, down, right right to reach (2, 2).

4 cherries were picked up during this single trip, and the matrix becomes [[0,1,-1],[0,0,-1],[0,0,0]].

Then, the player went left, up, up, left to return home, picking up one more cherry.

The total number of cherries picked up is 5, and this is the maximum possible.

Example 2:

Input: grid = [[1,1,-1],[1,-1,1],[-1,1,1]]
Output: 0

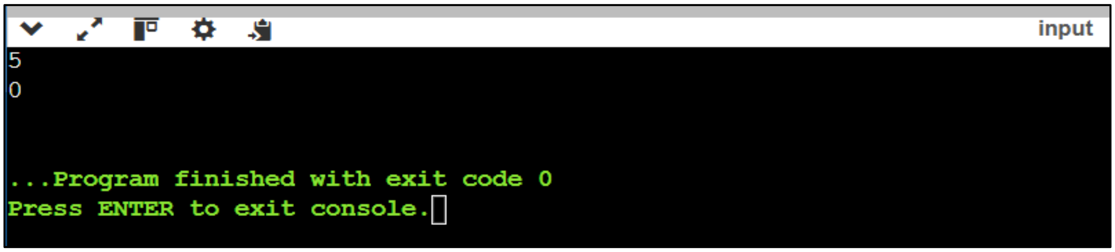
Constraints:

n == grid.length
n == grid[i].length
1 <= n <= 50
grid[i][j] is -1, 0, or 1.
grid[0][0] != -1
grid[n - 1][n - 1] != -1

Implementation/Code:

```
#include <iostream>
#include <vector>
#include <climits>
#include <algorithm>

using namespace std;

class Solution {
public:
    int cherryPickup(vector<vector<int>>& grid) {
        int n = grid.size();
        vector<vector<vector<int>>> dp(n, vector<vector<int>>(n, vector<int>(n, INT_MIN)));
        dp[0][0][0] = grid[0][0];

        for (int x1 = 0; x1 < n; x1++) {
            for (int y1 = 0; y1 < n; y1++) {
                for (int x2 = 0; x2 < n; x2++) {
                    int y2 = x1 + y1 - x2; // derived from x1 + y1 == x2 + y2
                    if (y2 < 0 || y2 >= n || grid[x1][y1] == -1 || grid[x2][y2] == -1) continue;
                    int cherries = grid[x1][y1];
                    if (x1 != x2) cherries += grid[x2][y2];

                    int maxPrev = INT_MIN;
                    if (x1 > 0 && x2 > 0) maxPrev = max(maxPrev, dp[x1 - 1][y1][x2 - 1]);
                    if (x1 > 0 && y2 > 0) maxPrev = max(maxPrev, dp[x1 - 1][y1][x2]);
                    if (y1 > 0 && x2 > 0) maxPrev = max(maxPrev, dp[x1][y1 - 1][x2 - 1]);
                    if (y1 > 0 && y2 > 0) maxPrev = max(maxPrev, dp[x1][y1 - 1][x2]);

                    if (maxPrev != INT_MIN) dp[x1][y1][x2] = cherries + maxPrev;
                }
            }
        }

        return max(0, dp[n - 1][n - 1][n - 1]);
    }
};

int main() {
    vector<vector<int>> grid1 = {
        {0, 1, -1},
        {1, 0, -1},
        {1, 1, 1}
    };
    vector<vector<int>> grid2 = {
        {1, 1, -1},
        {1, -1, 1},
        {-1, 1, 1}
    };
    Solution sol;
    int result = sol.cherryPickup(grid1);
    cout << result << endl;

    result = sol.cherryPickup(grid2);
    cout << result << endl;
    return 0;
}
```

Output:



```
5
0


...Program finished with exit code 0
Press ENTER to exit console.
```