

Day - 8

Name – Ishita

Faculty Name- Er. Rajni Devi

UID- 22BCS15353

Date- 29 Dec ,2024

Section – 620-B

Ques 1:

N-th Tribonacci Number

The Tribonacci sequence T_n is defined as follows:

$T_0 = 0$, $T_1 = 1$, $T_2 = 1$, and $T_{n+3} = T_n + T_{n+1} + T_{n+2}$ for $n \geq 0$.

Given n , return the value of T_n .

Code :-

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

int tribonacci(int n) {
    if (n == 0) return 0;
    if (n == 1 || n == 2) return 1;

    int t0 = 0, t1 = 1, t2 = 1, tn = 0;
    for (int i = 3; i <= n; i++) {
        tn = t0 + t1 + t2; // Calculate the next Tribonacci number
        t0 = t1;           // Shift the values
        t1 = t2;
        t2 = tn;
    }
    return tn;
}

int main() {
    int n;
    cout << "Enter the value of n: ";
    cin >> n;
    int result = tribonacci(n);
    cout << "The " << n << "-th Tribonacci number is: " << result << endl;
    return 0;
}
```

Output:

Output

```
Enter the value of n: 4
The 4-th Tribonacci number is: 4
```

Ques 2:

Divisor Game

Alice and Bob take turns playing a game, with Alice starting first. Initially, there is a number n on the chalkboard. On each player's turn, that player makes a move consisting of:

Choosing any x with $0 < x < n$ and $n \% x == 0$.

Replacing the number n on the chalkboard with $n - x$.

Also, if a player cannot make a move, they lose the game.

Return true if and only if Alice wins the game, assuming both players play optimally.

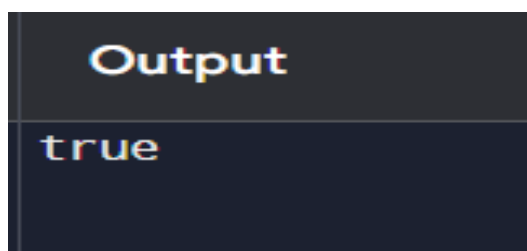
Code:

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

bool divisorGame(int n) {
    std::vector<bool> dp(n + 1, false); // dp[i] means if the position i is a winning position for
    the current player
    for (int i = 2; i <= n; ++i) {
        for (int x = 1; x < i; ++x) {
            if (i % x == 0 && !dp[i - x]) {
                dp[i] = true;
                break;}}
    }
    return dp[n]; // Alice wins if dp[n] is true
}

int main() {
    int n = 2;
    std::cout << (divisorGame(n) ? "true" : "false") << std::endl;
    return 0;
}
```

Output:



```
Output
true
```

Ques 3:

Pascal's Triangle II

Given an integer rowIndex, return the rowIndexth (0-indexed) row of the Pascal's triangle. In Pascal's triangle, each number is the sum of the two numbers directly above it as shown:

Code:

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

std::vector<int> getRow(int rowIndex) {
    std::vector<int> row(rowIndex + 1, 1); // Initialize the row with 1's
    for (int i = 1; i < rowIndex; ++i) {
        for (int j = i; j > 0; --j) {
            row[j] += row[j - 1];
        }
    }
    return row;
}

int main() {
    int rowIndex;
    std::cout << "Enter the row index: ";
    std::cin >> rowIndex;
    std::vector<int> result = getRow(rowIndex);
    std::cout << "Row " << rowIndex << " of Pascal's Triangle: ";
    for (int num : result) {
        std::cout << num << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

Output:

Output
Enter the row index: 3
Row 3 of Pascal's Triangle: 1 3 3 1

Ques 4:

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?

Code:

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

int climbStairs(int n) {
    if (n <= 1) return 1;

    std::vector<int> dp(n + 1, 0); // dp[i] will store the number of ways to reach the i-th step
    dp[0] = 1; // 1 way to be at the 0-th step (starting point)
    dp[1] = 1;
    for (int i = 2; i <= n; ++i) {
        dp[i] = dp[i - 1] + dp[i - 2];
    }
    return dp[n]; // The number of ways to reach the top (n-th step)
}

int main() {
    int n;

    std::cout << "Enter the number of steps: ";
    std::cin >> n;

    std::cout << "Number of ways to climb the stairs: " << climbStairs(n) << std::endl;

    return 0;
}
```

Output:

```
Output
Enter the number of steps: 2
Number of ways to climb the stairs: 2
```

Ques 5:

Counting Bits

Given an integer n , return an array `ans` of length $n + 1$ such that for each i ($0 \leq i \leq n$), `ans[i]` is the number of 1's in the binary representation of i .

Code:

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

std::vector<int> countBits(int n) {
    std::vector<int> ans(n + 1, 0); // Initialize the result array with 0's
    for (int i = 1; i <= n; ++i) {
        ans[i] = ans[i >> 1] + (i & 1); // Using the formula: ans[i] = ans[i / 2] + (i % 2)
    }
    return ans;}

int main() {
    int n;
    std::cout << "Enter the value of n: ";
    std::cin >> n;
    std::vector<int> result = countBits(n);
    std::cout << "Number of 1's in the binary representations: ";
    for (int num : result) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Output:

Output

```
Enter the value of n: 2
Number of 1's in the binary representations: 0 1 1
```

Ques 6:

Longest Palindromic Substring

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

Code

```
#include <iostream>
#include <string>

std::string longestPalindrome(std::string s) {
    if (s.empty()) return "";
    int start = 0, maxLength = 1;
    auto expandAroundCenter = [&](int left, int right) {
        while (left >= 0 && right < s.size() && s[left] == s[right]) {
            left--;
            right++;
        }
        return right - left - 1; // Return the length of the palindrome
    };
    for (int i = 0; i < s.size(); ++i) {
        // Odd-length palindrome
        int len1 = expandAroundCenter(i, i);
        // Even-length palindrome
        int len2 = expandAroundCenter(i, i + 1);

        // Choose the maximum length between the two
        int len = std::max(len1, len2);
        if (len > maxLength) {
            maxLength = len;
            start = i - (maxLength - 1) / 2;
        }
    }
    return s.substr(start, maxLength);
}
```

```

int main() {
    std::string s;
    std::cout << "Enter a string: ";
    std::cin >> s;
    std::cout << "Longest Palindromic Substring: " << longestPalindrome(s) << std::endl;
    return 0;
}

```

Output:

```

Output
Enter a string: babad
Longest Palindromic Substring: bab

```

Ques 7:

Generate Parentheses

Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

Code:

```

#include <iostream>

#include <vector>

#include <string>

void generateParenthesesHelper(int n, int open, int close, std::string current,
std::vector<std::string>& result) {

    if (current.length() == 2 * n) {

        result.push_back(current); // If the current string has 2 * n characters, add it to the result

        return;

    if (open < n) {

        generateParenthesesHelper(n, open + 1, close, current + "(", result);

    }
}

```

```

        if (close < open) {

            generateParenthesesHelper(n, open, close + 1, current + ")", result); // Add a close
            parenthesis
        }
    }

std::vector<std::string> generateParentheses(int n) {

    std::vector<std::string> result;

    generateParenthesesHelper(n, 0, 0, "", result);

    return result;
}

int main() {

    int n;

    std::cout << "Enter the number of pairs of parentheses: ";

    std::cin >> n;

    std::vector<std::string> result = generateParentheses(n);

    std::cout << "Generated parentheses combinations: ";

    for (const std::string& str : result) {

        std::cout << str << " ";

    }

    std::cout << std::endl;

    return 0;
}

```

Output:

```

Output

Enter the number of pairs of parentheses: 2
Generated parentheses combinations: (()) ()()

```


Ques 8:

Number of Digit One

Given an integer n, count the total number of digit 1 appearing in all non-negative integers less than or equal to n.

Code:

```
#include <iostream>

int countDigitOne(int n) {
    int count = 0;
    long long factor = 1;
    while (factor <= n) {
        int lowerNumbers = n - (n / factor) * factor;
        int currentDigit = (n / factor) % 10;
        int higherNumbers = n / (factor * 10);

        // Count based on the current digit position
        if (currentDigit == 0) {
            count += higherNumbers * factor;
        } else if (currentDigit == 1) {
            count += higherNumbers * factor + lowerNumbers + 1;
        } else {
            count += (higherNumbers + 1) * factor;
        }
        factor *= 10;
    }
    return count;
}

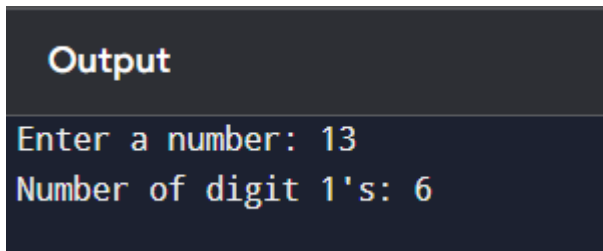
int main() {
    int n;
    std::cout << "Enter a number: ";
    std::cin >> n;
```

```

std::cout << "Number of digit 1's: " << countDigitOne(n) << std::endl;
return 0;
}

```

Output:



```

Output
Enter a number: 13
Number of digit 1's: 6

```

Ques 9:

Race Car

Your car starts at position 0 and speed +1 on an infinite number line. Your car can go into negative positions. Your car drives automatically according to a sequence of instructions 'A' (accelerate) and 'R' (reverse):

When you get an instruction 'A', your car does the following:

position += speed

speed *= 2

When you get an instruction 'R', your car does the following:

If your speed is positive then speed = -1

otherwise speed = 1

Code:

```
#include <iostream>
```

```
#include <queue>
```

```
#include <unordered_set>
```

```
#include <string>
```

```
int racecar(int target) {
```

```
    std::queue<std::pair<int, int>> q; // queue to store (position, speed)
```

```
    std::unordered_set<std::string> visited; // to track visited states (position, speed)
```

```

q.push({0, 1});

visited.insert("0,1");

int steps = 0;

while (!q.empty()) {
    int size = q.size();

    for (int i = 0; i < size; ++i) {
        auto [pos, speed] = q.front();

        q.pop();

        if (pos == target) return steps;

        int new_pos = pos + speed;

        int new_speed = speed * 2;

        std::string new_state = std::to_string(new_pos) + "," + std::to_string(new_speed);

        if (visited.find(new_state) == visited.end()) {
            visited.insert(new_state);

            q.push({new_pos, new_speed});
        }

        new_speed = (speed > 0) ? -1 : 1;

        new_state = std::to_string(pos) + "," + std::to_string(new_speed);

        if (visited.find(new_state) == visited.end()) {
            visited.insert(new_state);

            q.push({pos, new_speed});
        }
    }

    ++steps; // Increment steps after processing all current states
}

return -1; // If no solution found, although it should always find one
}

```

```

int main() {
    int target;

    std::cout << "Enter the target position: ";

    std::cin >> target;

    std::cout << "Shortest sequence of instructions to reach the target: " << racecar(target) ;

    return 0;
}

```

Output:

```

Output

Enter the target position: 3
Shortest sequence of instructions to reach the target: 2

```

Ques 10:

Super Egg Drop

You are given k identical eggs and you have access to a building with n floors labeled from 1 to n . You know that there exists a floor f where $0 \leq f \leq n$ such that any egg dropped at a floor higher than f will break, and any egg dropped at or below floor f will not break. Each move, you may take an unbroken egg and drop it from any floor x (where $1 \leq x \leq n$). If the egg breaks, you can no longer use it. However, if the egg does not break, you may reuse it in future moves. Return the minimum number of moves that you need to determine with certainty what the value of f is.

Code:

```

#include <iostream>

#include <vector>

#include <algorithm>

int superEggDrop(int k, int n) {
    // dp[i][j] will represent the maximum number of floors we can check with i eggs and j moves.

    std::vector<std::vector<int>> dp(k + 1, std::vector<int>(n + 1, 0));

```

```

// Fill the dp table for all the egg count scenarios.
for (int i = 1; i <= k; ++i) {
    for (int j = 1; j <= n; ++j) {
        // The relationship is:  $dp[i][j] = dp[i - 1][j - 1] + dp[i][j - 1] + 1$ 
        dp[i][j] = dp[i - 1][j - 1] + dp[i][j - 1] + 1;
        if (dp[i][j] >= n) {
            return j;
        }
    }
}
return n;
}

int main() {
    int k, n;
    std::cout << "Enter the number of eggs: ";
    std::cin >> k;
    std::cout << "Enter the number of floors: ";
    std::cin >> n;
    std::cout << "Minimum number of moves required: " << superEggDrop(k, n) << std::endl;
    return 0;
}

```

Output:

Output

```

Enter the number of eggs: 1
Enter the number of floors: 2
Minimum number of moves required: 2

```