

## DOMAIN WINTER WINNING CAMP

DAY: 8

Submitted by: Agrima Sharma

UID: 22BCS15314

Section: 620/A

Very Easy

### 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) {
    // Base cases
    if (n == 0) return 0;
    if (n == 1 || n == 2) return 1;

    // Variables to store  $T_{n-3}$ ,  $T_{n-2}$ ,  $T_{n-1}$ , and  $T_n$ 
    int t0 = 0, t1 = 1, t2 = 1, t3;

    // Iteratively compute the next Tribonacci number
    for (int i = 3; i <= n; ++i) {
        t3 = t0 + t1 + t2; //  $T_n = T_{n-1} + T_{n-2} + T_{n-3}$ 
        t0 = t1; // Update t0 to t1
        t1 = t2; // Update t1 to t2
        t2 = t3; // Update t2 to the newly computed t3
    }

    return t2; // Return  $T_n$ 
}

int main() {
```

```

int n;
cout << "Enter n: ";
cin >> n;

cout << "The " << n << "-th Tribonacci number is: " << tribonacci(n) << endl;
return 0;
}

```

### Output

```

Enter n: 5
The 5-th Tribonacci number is: 7

```

## 2. DivisorGame

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>
using namespace std;

bool divisorGame(int n) {
    // Alice wins if n is even, Bob wins if n is odd
    return n % 2 == 0;
}

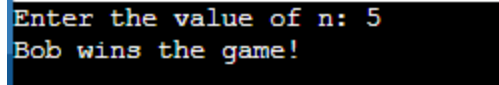
int main() {
    int n;
    cout << "Enter the value of n: ";
    cin >> n;

    if (divisorGame(n)) {
        cout << "Alice wins the game!" << endl;
    } else {
        cout << "Bob wins the game!" << endl;
    }
}

```

```
    return 0;
}
```

**Output:**



```
Enter the value of n: 5
Bob wins the game!
```

### 3. MaximumRepeatingSubstring

For a string sequence, a string word is k-repeating if word concatenated k times is a substring of sequence. The word's maximum k-repeating value is the highest value k where word is k-repeating in sequence. If word is not a substring of sequence, word's maximum k-repeating value is 0. Given strings sequence and word, return the maximum k-repeating value of word in sequence.

**Code:**

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

int maxRepeating(string sequence, string word) {
    int k = 0;
    string repeatedWord = word;

    // Keep repeating the word and check if it's a substring of sequence
    while (sequence.find(repeatedWord) != string::npos) {
        k++;
        repeatedWord += word; // Concatenate word one more time
    }

    return k;
}

int main() {
    string sequence, word;
    cout << "Enter the sequence: ";
    cin >> sequence;
    cout << "Enter the word: ";
    cin >> word;
```

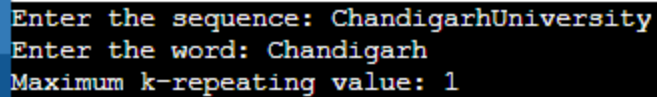
```

int result = maxRepeating(sequence, word);
cout << "Maximum k-repeating value: " << result << endl;

return 0;
}

```

**Output:**



```

Enter the sequence: ChandigarhUniversity
Enter the word: Chandigarh
Maximum k-repeating value: 1

```

#### 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>
using namespace std;

int climbStairs(int n) {
    if (n == 0) return 0; // No steps to climb
    if (n == 1) return 1; // Only one way to climb (1 step)

    int prev2 = 1, prev1 = 1, current;

    for (int i = 2; i <= n; ++i) {
        current = prev1 + prev2; // current = number of ways to reach step i
        prev2 = prev1; // Update prev2 to prev1 (previous step)
        prev1 = current; // Update prev1 to current (current step)
    }

    return prev1; // Return the number of ways to reach the nth step
}

int main() {
    int n;
    cout << "Enter the number of steps: ";
    cin >> n;
}

```

```

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

    return 0;
}

```

**Output:**

```

Enter the number of steps: 5
Number of distinct ways to climb to the top: 8

```

## 5. BestTimeToBuyandSellStock

You are given an array prices where prices[i] is the price of a given stock on the ith day. You want to maximize your profit by choosing a single day to buy one stock and choosing a different day in the future to sell that stock. Return the maximum profit you can achieve from this transaction. If you cannot achieve any profit, return 0.

**Code:**

```

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

int maxProfit(vector<int>& prices) {
    int min_price = INT_MAX; // Start with a very large value
    int max_profit = 0;      // Initialize max profit as 0

    // Traverse through the list of prices
    for (int price : prices) {
        // Update the minimum price encountered so far
        min_price = min(min_price, price);

        // Calculate the profit if selling at the current price
        int profit = price - min_price;

        // Update the maximum profit if we found a new higher profit
        max_profit = max(max_profit, profit);
    }

    return max_profit; // Return the maximum profit found
}

```

```

}

int main() {
    vector<int> prices;
    int n, price;

    // Read the number of days (prices array length)
    cout << "Enter the number of days: ";
    cin >> n;

    // Read the prices for each day
    cout << "Enter the prices for each day: ";
    for (int i = 0; i < n; i++) {
        cin >> price;
        prices.push_back(price);
    }

    // Call the function to get the maximum profit
    int result = maxProfit(prices);

    cout << "Maximum profit: " << result << endl;

    return 0;
}

```

**Output:**

```

Enter the number of days: 5
Enter the prices for each day: 100
200
100
100
500
Maximum profit: 400

```

## 6. 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>
using namespace std;

```

```

vector<int> countBits(int n) {
    vector<int> ans(n + 1, 0); // Initialize the result array with size n + 1 and 0s

    for (int i = 1; i <= n; ++i) {
        ans[i] = ans[i / 2] + (i % 2); // Use the recurrence relation
    }

    return ans;
}

int main() {
    int n;
    cout << "Enter the value of n: ";
    cin >> n;

    vector<int> result = countBits(n);

    cout << "The number of 1's in the binary representation of numbers from 0 to " <<
n << " is: ";
    for (int i = 0; i <= n; ++i) {
        cout << result[i] << " ";
    }
    cout << endl;

    return 0;
}

```

**Output:**

```

Enter the value of n: 5
The number of 1's in the binary representation of numbers from 0 to 5 is: 0 1 1 2 1 2

```

## 7. IsSubsequence

Given two strings *s* and *t*, return true if *s* is a subsequence of *t*, or false otherwise. A subsequence of a string is a new string that is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (i.e., "ace" is a subsequence of "abcde" while "aec" is not).

**Code:**

```

#include <iostream>
#include <string>

```

```

using namespace std;

bool isSubsequence(string s, string t) {
    int sIndex = 0, tIndex = 0;

    // Traverse through both strings
    while (sIndex < s.size() && tIndex < t.size()) {
        if (s[sIndex] == t[tIndex]) {
            sIndex++; // Move pointer for s forward
        }
        tIndex++; // Always move pointer for t forward
    }

    // If all characters in s have been matched
    return sIndex == s.size();
}

int main() {
    string s, t;

    cout << "Enter string s: ";
    cin >> s;
    cout << "Enter string t: ";
    cin >> t;

    if (isSubsequence(s, t)) {
        cout << "\"" << s << "\" is a subsequence of \"" << t << "\".\n";
    } else {
        cout << "\"" << s << "\" is NOT a subsequence of \"" << t << "\".\n";
    }

    return 0;
}

```

**Output:**

```

Enter string s: Chandigarh
Enter string t: University
"Chandigarh" is NOT a subsequence of "University".

```



## 8. Longest Palindromic Substring

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

**Code:**

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

string longestPalindrome(string s) {
    if (s.empty()) return "";

    int start = 0, maxLength = 1;

    // Helper function to expand around the center
    auto expandAroundCenter = [&](int left, int right) {
        while (left >= 0 && right < s.size() && s[left] == s[right]) {
            left--;
            right++;
        }
        return right - left - 1; // Length of the current palindrome
    };

    for (int i = 0; i < s.size(); ++i) {
        // Odd-length palindromes (single character center)
        int len1 = expandAroundCenter(i, i);
        // Even-length palindromes (pair of characters center)
        int len2 = expandAroundCenter(i, i + 1);

        // Get the maximum length of both palindrome types
        int len = max(len1, len2);

        // If we found a longer palindrome, update start and maxLength
        if (len > maxLength) {
            maxLength = len;
            start = i - (maxLength - 1) / 2;
        }
    }

    // Return the longest palindromic substring
```

```

        return s.substr(start, maxLength);
    }

    int main() {
        string s;
        cout << "Enter the string: ";
        cin >> s;

        string result = longestPalindrome(s);
        cout << "Longest palindromic substring: " << result << endl;

        return 0;
    }

```

**Output:**

```

Enter the string: aabcddea
Longest palindromic substring: aa

```

## 9. GenerateParentheses

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

**Code:**

```

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

```

```

void generateParenthesesHelper(int open_count, int close_count, int n, string
current, vector<string>& result) {

```

```

    // If the current string has used n opening and n closing parentheses, it's a valid
combination

```

```

    if (open_count == n && close_count == n) {
        result.push_back(current);
        return;
    }

```

```

    // If we can add an opening parenthesis, do so

```

```

        if (open_count < n) {
            generateParenthesesHelper(open_count + 1, close_count, n, current + "(",
result);
        }

        // If we can add a closing parenthesis, do so
        if (close_count < open_count) {
            generateParenthesesHelper(open_count, close_count + 1, n, current + ")",
result);
        }
    }
}

vector<string> generateParentheses(int n) {
    vector<string> result;
    generateParenthesesHelper(0, 0, n, "", result);
    return result;
}

int main() {
    int n;
    cout << "Enter the number of pairs of parentheses: ";
    cin >> n;

    vector<string> result = generateParentheses(n);

    cout << "Generated well-formed parentheses combinations:" << endl;
    for (const string& s : result) {
        cout << s << endl;
    }

    return 0;
}

```

**Output:**

```

1  Enter the number of pairs of parentheses: 2
    Generated well-formed parentheses combinations:
    (())
    ()()

```

## 10. 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.**

**Code:**

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

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); // Array to store the heights of columns
    int maxArea = 0;

    // Iterate through each row in the matrix
    for (int i = 0; i < rows; ++i) {
        // Update the heights for the histogram based on the current row
        for (int j = 0; j < cols; ++j) {
            // If the current cell is '1', add to the height, else reset to 0
            heights[j] = matrix[i][j] == '1' ? heights[j] + 1 : 0;
        }

        // Now, we need to find the largest rectangle in this histogram (heights)
        stack<int> st;
        for (int j = 0; j <= cols; ++j) {
            // We use an extra element to ensure we handle the remaining elements in
            // the stack
            int h = (j == cols) ? 0 : heights[j];

            // Calculate the area for the histogram bars stored in the stack
            while (!st.empty() && h < heights[st.top()]) {
                int height = heights[st.top()];
                st.pop();
                int width = st.empty() ? j : j - st.top() - 1;
                maxArea = max(maxArea, height * width);
            }
            st.push(j);
        }
    }

    return maxArea;
}
```

```

    }

    st.push(j);
}
}

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'}
    };

    cout << "Maximal Rectangle Area: " << maximalRectangle(matrix) << endl;

    return 0;
}

```

**Output:**

```
Maximal Rectangle Area: 6
```