DOMAIN WINTER WINNING CAMP ASSIGNMENT

Student Name: Vaibhav Chhillar   **UID: 22BCS12585**
Branch: BE-CSE::CS201   **Section/Group: 22BCS_FL_IOT-603/A**
Semester: 5th

➢ DAY-3 [21-12-2024]

1. **Fibonacci Series Using Recursion**   *(Very Easy)*

The Fibonacci numbers, commonly denoted F(n) form a sequence, called the Fibonacci sequence, such that each number is the sum of the two preceding ones, starting from 0 and 1. That is,

 F(0) = 0, F(1) = 1

F(n) = F(n - 1) + F(n - 2), for n > 1.

Given n, calculate F(n).

Example 1:

Input: n = 2

Output: 1

Explanation: F(2) = F(1) + F(0) = 1 + 0 = 1

Example 2:

Input: n = 3

Output: 2

Explanation: F(3) = F(2) + F(1) = 1 + 1 = 2.

Constraints:

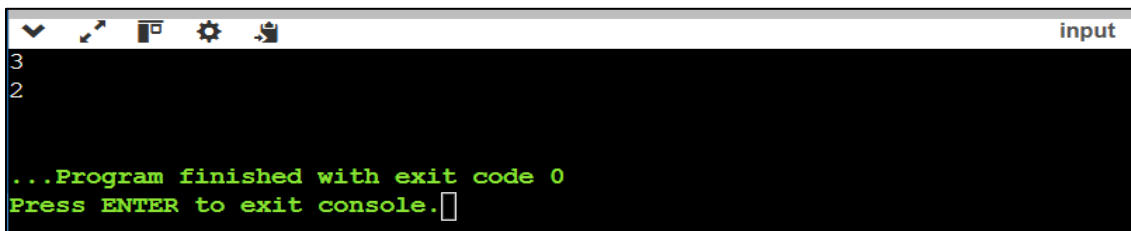0 <= n <= 30

Implementation/Code:

```cpp
#include <iostream>                    //Programming in C++
using namespace std;

int fibonacci(int n) {
    if (n <= 1) return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}

int main() {
    int n;
    cin >> n;
    if(n>=0 && n<=30)
    {
        cout << fibonacci(n) << endl;
    }
    else
    {
        cout<<"Invalid input.";
        return 1;
    }
    return 0;
}
```

Output:


```
3
2

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

2. Reverse Linked List   *(Easy)*

Given the head of a singly linked list, reverse the list, and return the reversed list.

Example 1:
Input: head = [1,2,3,4,5]
Output: [5,4,3,2,1]

Example 2:
Input: head = [1,2]
Output: [2,1]

Constraints:
The number of nodes in the list is the range [0, 5000].
-5000 <= Node.val <= 5000

Follow up: A linked list can be reversed either iteratively or recursively. Could you implement both?

Implementation/Code:

```cpp
#include <iostream>                                    //Programming in C++
using namespace std;

// Definition for singly-linked list.
struct ListNode {
    int val;
    ListNode* next;
    ListNode(int x) : val(x), next(nullptr) {}
};

// Iterative approach to reverse the linked list
ListNode* reverseListIterative(ListNode* head) {
    ListNode* prev = nullptr;
    ListNode* current = head;

    while (current != nullptr) {
        ListNode* nextNode = current->next; // Save the next node
        current->next = prev;               // Reverse the link
        prev = current;                     // Move prev to current
```

```cpp
        current = nextNode;          // Move current to next node
    }

    return prev; // New head of the reversed list
}

// Recursive approach to reverse the linked list
ListNode* reverseListRecursive(ListNode* head) {
    // Base case: If the list is empty or has one node
    if (head == nullptr || head->next == nullptr) {
        return head;
    }

    // Reverse the rest of the list
    ListNode* newHead = reverseListRecursive(head->next);

    // Adjust the pointers
    head->next->next = head;
    head->next = nullptr;

    return newHead;
}

// Helper function to print the linked list
void printList(ListNode* head) {
    cout<<"[";
    while (head != nullptr) {
        cout << head->val << ", ";
        head = head->next;
    }
    cout <<"]"<< endl;
}

// Main function to test the solution
int main() {
    // Example 1: Create the linked list [1, 2, 3, 4, 5]
    ListNode* head = new ListNode(1);
    head->next = new ListNode(2);
```

```cpp
    head->next->next = new ListNode(3);
    head->next->next->next = new ListNode(4);
    head->next->next->next->next = new ListNode(5);

    cout << "Original List: ";
    printList(head);

    // Test iterative reversal
    ListNode* reversedIterative = reverseListIterative(head);
    cout << "Reversed List (Iterative): ";
    printList(reversedIterative);

    // Reset the list for recursive test
    head = new ListNode(1);
    head->next = new ListNode(2);
    head->next->next = new ListNode(3);
    head->next->next->next = new ListNode(4);
    head->next->next->next->next = new ListNode(5);

    // Test recursive reversal
    ListNode* reversedRecursive = reverseListRecursive(head);
    cout << "Reversed List (Recursive): ";
    printList(reversedRecursive);

    return 0;
}
```
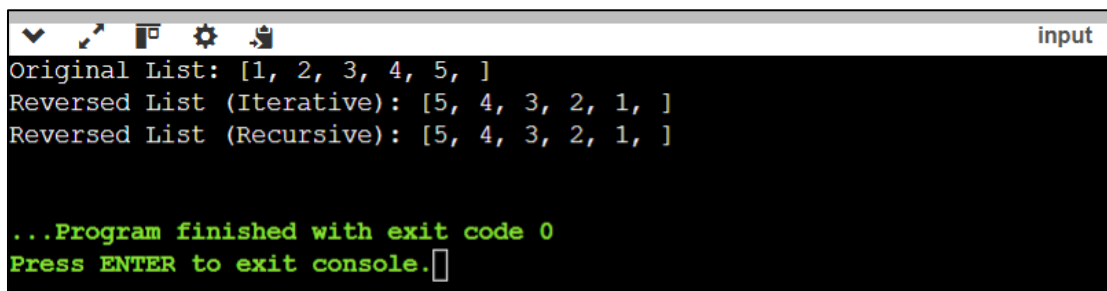
Output:

3. Add Two Numbers   *(Medium)*

You are given two non-empty linked lists representing two non-negative integers. The digits are stored in reverse order, and each of their nodes contains a single digit. Add the two numbers and return the sum as a linked list.
You may assume the two numbers do not contain any leading zero, except the number 0 itself.

Example 1:
Input: l1 = [2,4,3], l2 = [5,6,4]
Output: [7,0,8]
Explanation: 342 + 465 = 807.
Example 2:
 Input: l1 = [0], l2 = [0]
Output: [0]
Example 3:
Input: l1 = [9,9,9,9,9,9,9], l2 = [9,9,9,9]
Output: [8,9,9,9,0,0,0,1]

Constraints:
 The number of nodes in each linked list is in the range [1, 100].
0 <= Node.val <= 9
It is guaranteed that the list represents a number that does not have leading zeros.

Implementation/Code:

```cpp
#include <iostream>                                    //Programming in C++
using namespace std;

struct ListNode {
    int val;
    ListNode* next;
    ListNode(int x) : val(x), next(nullptr) {}
};

ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
    ListNode* dummy = new ListNode(0);
```

```cpp
        ListNode* current = dummy;
        int carry = 0;
        while (l1 || l2 || carry) {
            int sum = carry;
            if (l1) {
                sum += l1->val;
                l1 = l1->next;
            }
            if (l2) {
                sum += l2->val;
                l2 = l2->next;
            }
            carry = sum / 10;
            current->next = new ListNode(sum % 10);
            current = current->next;
        }
        return dummy->next;
}

int main() {
    int n1, n2, val;
    cin >> n1;
    ListNode* l1 = nullptr;
    ListNode* tail1 = nullptr;
    for (int i = 0; i < n1; ++i) {
        cin >> val;
        ListNode* newNode = new ListNode(val);
        if (!l1) l1 = tail1 = newNode;
        else {
            tail1->next = newNode;
            tail1 = newNode;
        }
    }

    cin >> n2;
    ListNode* l2 = nullptr;
    ListNode* tail2 = nullptr;
    for (int i = 0; i < n2; ++i) {
```
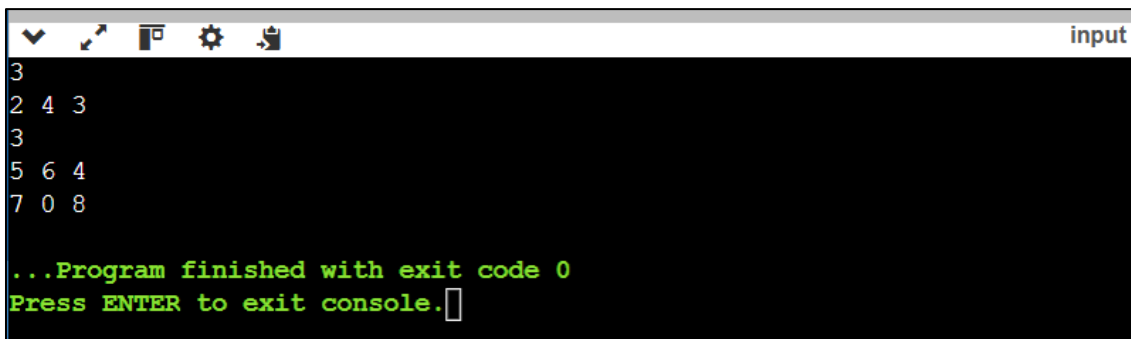
```
        cin >> val;
        ListNode* newNode = new ListNode(val);
        if (!l2) l2 = tail2 = newNode;
        else {
            tail2->next = newNode;
            tail2 = newNode;
        }
    }

    ListNode* result = addTwoNumbers(l1, l2);
    while (result) {
        cout << result->val << " ";
        result = result->next;
    }
    return 0;
}
```

Output:

4. Wildcard Matching   *(Hard)*

Given an input string (s) and a pattern (p), implement wildcard pattern matching with support for '?' and '*' where:

'?' Matches any single character.
'*' Matches any sequence of characters (including the empty sequence).
The matching should cover the entire input string (not partial).

Example 1:

Input: s = "aa", p = "a"

Output: false

Explanation: "a" does not match the entire string "aa".

Example 2:

Input: s = "aa", p = "*"

Output: true

Explanation: '*' matches any sequence.

Example 3:

Input: s = "cb", p = "?a"

Output: false

Explanation: '?' matches 'c', but the second letter is 'a', which does not match 'b'.

Constraints:

0 <= s.length, p.length <= 2000

s contains only lowercase English letters.

p contains only lowercase English letters, '?' or '*'.

Implementation/Code:

```cpp
#include <iostream>                                    //Programming in C++
#include <vector>
using namespace std;

bool isMatch(string s, string p) {
    int m = s.size(), n = p.size();
    vector<vector<bool>> dp(m + 1, vector<bool>(n + 1, false));
    dp[0][0] = true;

    for (int j = 1; j <= n; ++j)
        if (p[j - 1] == '*') dp[0][j] = dp[0][j - 1];

    for (int i = 1; i <= m; ++i) {
        for (int j = 1; j <= n; ++j) {
            if (p[j - 1] == '?' || p[j - 1] == s[i - 1])
                dp[i][j] = dp[i - 1][j - 1];
```

```
        else if (p[j - 1] == '*')
            dp[i][j] = dp[i - 1][j] || dp[i][j - 1];
    }
  }
  return dp[m][n];
}

int main() {
    string s, p;
    cin >> s >> p;
    cout << (isMatch(s, p) ? "true" : "false") << endl;
    return 0;
}
```

Output:

```
                                                    input
aa
a
false


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

5. Special Binary String   *(Very Hard)*

Special binary strings are binary strings with the following two properties:
The number of 0's is equal to the number of 1's.
Every prefix of the binary string has at least as many 1's as 0's.

You are given a special binary string s.
A move consists of choosing two consecutive, non-empty, special substrings of s, and swapping them. Two strings are consecutive if the last character of the first string is exactly one index before the first character of the second string.

Return the lexicographically largest resulting string possible after applying the mentioned operations on the string.

Example 1:
Input: s = "11011000"
Output: "11100100"
Explanation: The strings "10" [occuring at s[1]] and "1100" [at s[3]] are swapped.
This is the lexicographically largest string possible after some number of swaps.

Example 2:
Input: s = "10"
Output: "10"

Constraints:
1 <= s.length <= 50
s[i] is either '0' or '1'.
s is a special binary string.

Implementation/Code:

```cpp
#include <iostream>                                    //Programming in C++
#include <string>
#include <vector>
#include <algorithm>
#include <numeric> // Include this header for accumulate
using namespace std;

string makeLargestSpecial(string s) {
    vector<string> specials;
    int count = 0, start = 0;
    for (int i = 0; i < s.size(); ++i) {
        count += (s[i] == '1') ? 1 : -1;
        if (count == 0) {
            specials.push_back("1" + makeLargestSpecial(s.substr(start + 1, i - start - 1)) +
"0");
            start = i + 1;
        }
    }
    sort(specials.rbegin(), specials.rend());
    return accumulate(specials.begin(), specials.end(), string(""));
}
```

```
int main() {
    string s;
    cin >> s;
    cout << makeLargestSpecial(s) << endl;
    return 0;
}
```

Output:


```
11011000
11100100


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

6. Printing first name and last name using function   *(Very Easy)*

You are given the firstname and lastname of a person on two different lines. Your task is to read them and print the following using function:

Hello firstname lastname! You just delved into function.

Function Description:
Complete the print_full_name function in the editor below.
print_full_name has the following parameters:
string first: the first name
string last: the last name

Prints
string: 'Hello firstname lastname   ! You just delved into using the function' where firstname and lastname are replaced with first and last.

Input Format
The first line contains the first name, and the second line contains the last name.

Constraints

The length of the first and last names are each ≤10 .

Sample Input 0

Ross

Taylor

Sample Output 0

Hello Ross Taylor! You just delved into function

Explanation 0

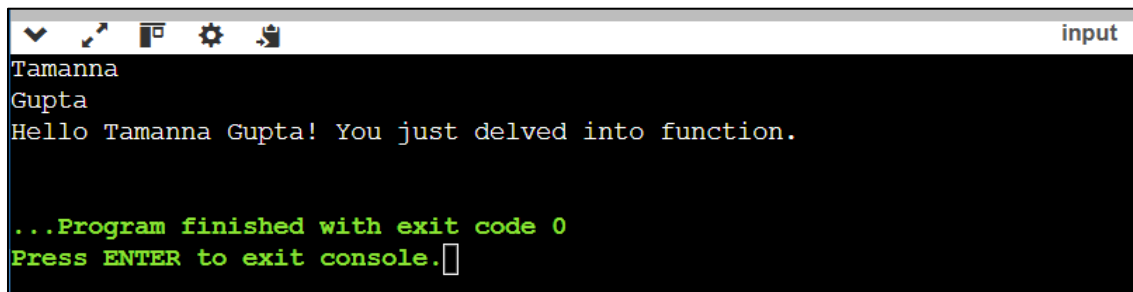The input read by the program is stored as a string data type. A string is a collection of characters.

Implementation/Code:

```cpp
#include <iostream>                                    //Programming in C++
using namespace std;

void print_full_name(string first, string last) {
    cout << "Hello " << first << " " << last << "! You just delved into function." << endl;
}

int main() {
    string first, last;
    cin >> first >> last;
    print_full_name(first, last);
    return 0;
}
```

Output:

```
Tamanna
Gupta
Hello Tamanna Gupta! You just delved into function.


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

7. Find GCD of Number using Function   *(Easy)*

Given an integer array nums, return the greatest common divisor of the smallest number and largest number in nums. The greatest common divisor of two numbers is the largest positive integer that evenly divides both numbers.

Example 1:

Input: nums = [2,5,6,9,10]
Output: 2
Explanation:
The smallest number in nums is 2.
The largest number in nums is 10.
The greatest common divisor of 2 and 10 is 2.

Example 2:

Input: nums = [7,5,6,8,3]
Output: 1
Explanation:
The smallest number in nums is 3.
The largest number in nums is 8.
The greatest common divisor of 3 and 8 is 1.

Example 3:

Input: nums = [3,3]
Output: 3
Explanation:
The smallest number in nums is 3.
The largest number in nums is 3.
The greatest common divisor of 3 and 3 is 3.

Constraints:
2 <= nums.length <= 1000
1 <= nums[i] <= 1000

Implementation/Code:

```cpp
#include <iostream>                                    //Programming in C++
#include <vector>
#include <algorithm>
using namespace std;

int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a % b);
}

int findGCD(vector<int>& nums) {
    int smallest = *min_element(nums.begin(), nums.end());
    int largest = *max_element(nums.begin(), nums.end());
    return gcd(smallest, largest);
}

int main() {
    int n, num;
    cin >> n;
    vector<int> nums;
    for (int i = 0; i < n; ++i) {
        cin >> num;
        nums.push_back(num);
    }
    cout << findGCD(nums) << endl;
    return 0;
}
```

Output:

```
5
2 5 6 9 10
2


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

8. Longest Substring Without Repeating Characters *(Medium)*

You are working on building a text editor application. One of the features you're focusing on is a "word suggestion" system. The editor needs to identify the longest sequence of characters typed without repeating any letters to suggest potential words or phrases. To accomplish this, you must efficiently find the length of the longest substring of unique characters as the user types.

Description:
Write a function that takes a string as input and returns the length of the longest substring without repeating characters. A substring is a contiguous sequence of characters within the string.

Example 1:
Input: "abcabcbb"
Output: 3
Explanation: The longest substring without repeating characters is "abc", which has length 3.

Constraints:
- The input string will have a length between 111 and 10410^4104.
- The characters in the string are printable ASCII characters.

This problem is a classic example of the **sliding window** technique.

Implementation/Code:

```cpp
#include <iostream>                                    //Programming in C++
#include <unordered_map>
using namespace std;

int lengthOfLongestSubstring(string s) {
    unordered_map<char, int> charIndex;
    int maxLength = 0, start = 0;

    for (int i = 0; i < s.size(); ++i) {
```
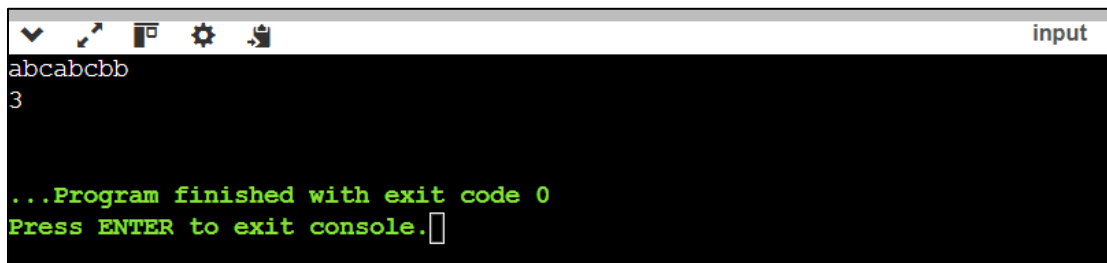
```
            if (charIndex.find(s[i]) != charIndex.end() && charIndex[s[i]] >= start) {
                start = charIndex[s[i]] + 1;
            }
            charIndex[s[i]] = i;
            maxLength = max(maxLength, i - start + 1);
        }
        return maxLength;
}

int main() {
    string s;
    cin >> s;
    cout << lengthOfLongestSubstring(s) << endl;
    return 0;
}
```

Output:

9. Maximum Subarray Product   *(Hard)*

You are developing a financial analysis application where users input daily stock price changes in an array. Your task is to determine the maximum profit or loss achievable over a contiguous period, considering that the profit/loss for a period is calculated by multiplying the daily percentage changes. This problem is critical for finding optimal periods to make decisions for investments or short-term trading.

Description:
Write a function that takes an integer array as input and returns the maximum product of a contiguous subarray. The array can contain both positive and negative integers, and your function must account for these to find the optimal subarray.

Input:
- An array of integers nums[]nums[]nums[] where 1≤nums.length≤1041 \leq nums.length \leq 10^41≤nums.length≤104 and −10≤nums[i]≤10-10 \leq nums[i] \leq 10−10≤nums[i]≤10.

Output:
- An integer representing the maximum product of any contiguous subarray.

Example 1:
Input: nums = [2, 3, -2, 4]
Output: 6
Explanation: The subarray [2, 3] has the largest product = 6.

Constraints:
1. The input array may contain both positive and negative numbers, including zero.
2. You must solve the problem with a time complexity of O(n).

Implementation/Code:

```cpp
#include <iostream>                              //Programming in C++
#include <vector>
#include <algorithm>
using namespace std;

int maxProduct(vector<int>& nums) {
    int maxProd = nums[0], minProd = nums[0], result = nums[0];

    for (int i = 1; i < nums.size(); ++i) {
        if (nums[i] < 0) swap(maxProd, minProd);
        maxProd = max(nums[i], maxProd * nums[i]);
        minProd = min(nums[i], minProd * nums[i]);
        result = max(result, maxProd);
    }
    return result;
}

int main() {
    int n, num;
```
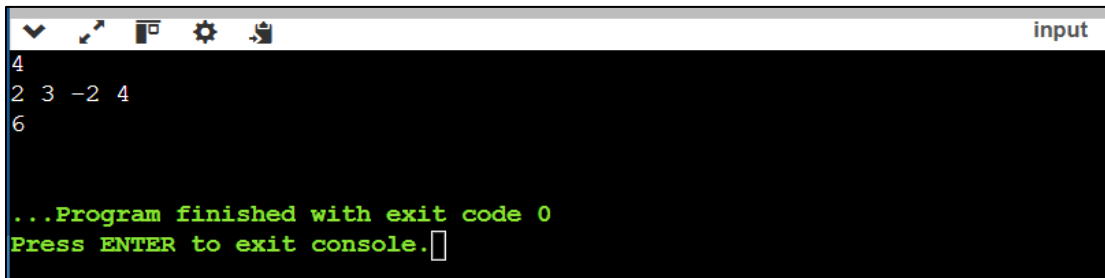
```
    cin >> n;
    vector<int> nums;
    for (int i = 0; i < n; ++i) {
        cin >> num;
        nums.push_back(num);
    }
    cout << maxProduct(nums) << endl;
    return 0;
}
```

Output:

10. Minimum Candies Distribution   *(Very Hard)*

There are n children standing in a line. Each child is assigned a rating value given in the integer array ratings.

You are giving candies to these children subjected to the following requirements:
Each child must have at least one candy.
Children with a higher rating get more candies than their neighbors.

Return the minimum number of candies you need to have to distribute the candies to the children.

Example 1:
Input: ratings = [1,0,2]
Output: 5
Explanation: You can allocate to the first, second and third child with 2, 1, 2 candies respectively.

Example 2:

Input: ratings = [1,2,2]

Output: 4

Explanation: You can allocate to the first, second and third child with 1, 2, 1 candies respectively.

The third child gets 1 candy because it satisfies the above two conditions.

Constraints:

n == ratings.length

$1 <= n <= 2 * 104$

$0 <= ratings[i] <= 2 * 104$

Implementation/Code:

```cpp
#include <iostream>                                    //Programming in C++
#include <vector>
#include <algorithm>
using namespace std;

int candy(vector<int>& ratings) {
    int n = ratings.size();
    vector<int> candies(n, 1);

    for (int i = 1; i < n; ++i) {
        if (ratings[i] > ratings[i - 1]) {
            candies[i] = candies[i - 1] + 1;
        }
    }

    for (int i = n - 2; i >= 0; --i) {
        if (ratings[i] > ratings[i + 1]) {
            candies[i] = max(candies[i], candies[i + 1] + 1);
        }
    }

    return accumulate(candies.begin(), candies.end(), 0);
}
```

```cpp
int main() {
    int n, rating;
    cin >> n;
    vector<int> ratings;
    for (int i = 0; i < n; ++i) {
        cin >> rating;
        ratings.push_back(rating);
    }
    cout << candy(ratings) << endl;
    return 0;
}
```

Output:

```
3
1 0 2
5


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