# WINTER WINNING DOMAIN CAMP 2024

# DAY 3 ASSIGNMENT

1)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

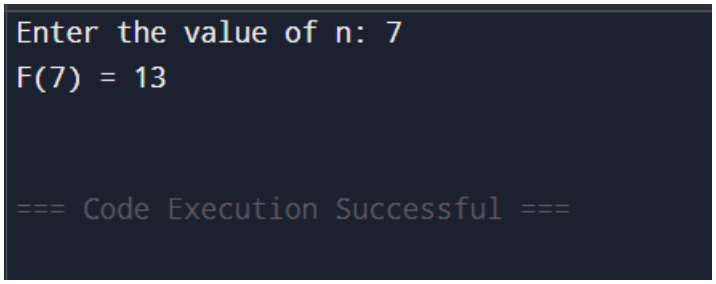
F(n) = F(n - 1) + F(n - 2), for n > 1.

Given n, calculate F(n).

**SOLUTION:**

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

int fibonacci(int n) {
    if (n == 0) return 0; // Base case: F(0) = 0
    if (n == 1) return 1; // Base case: F(1) = 1
    return fibonacci(n - 1) + fibonacci(n - 2); // Recursive case
}

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

    cout << "F(" << n << ") = " << fibonacci(n) << endl;
    return 0;
}
```

**OUTPUT:**




2)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

**SOLUTION:**

```
#include <iostream>
using namespace std;
struct ListNode {
    int val;
    ListNode* next;
    ListNode(int x) : val(x), next(nullptr) {}
};

class Solution {
```

```
public:
    ListNode* reverseList(ListNode* head) {
        // Base case: if head is null or only one node, return it
        if (head == nullptr || head->next == nullptr) {
            return head;
        }
        ListNode* reversedHead = reverseList(head->next);
        // Reverse the current node's link
        head->next->next = head;
        head->next = nullptr;
        return reversedHead;
    }
};
ListNode* createList(int arr[], int n) {
    if (n == 0) return nullptr;
    ListNode* head = new ListNode(arr[0]);
    ListNode* tail = head;
    for (int i = 1; i < n; i++) {
        tail->next = new ListNode(arr[i]);
        tail = tail->next;
    }
    return head;
}

void printList(ListNode* head) {
    while (head != nullptr) {
        cout << head->val;
        if (head->next) cout << " -> ";
        head = head->next;
    }
    cout << endl;
```

```
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    ListNode* head = createList(arr, n);
    cout << "Original List: ";
    printList(head);

    Solution sol;
    ListNode* reversedHead = sol.reverseList(head);

    cout << "Reversed List: ";
    printList(reversedHead);
    return 0;
}
```

OUTPUT:

```
Original List: 1 -> 2 -> 3 -> 4 -> 5
Reversed List: 5 -> 4 -> 3 -> 2 -> 1


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

3)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]

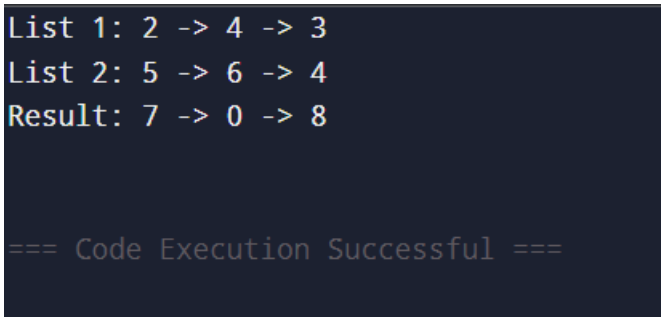
**SOLUTION:**

```
#include <iostream>
using namespace std;
struct ListNode {
    int val;
    ListNode* next;
    ListNode(int x) : val(x), next(nullptr) {}
};

class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        ListNode* dummyHead = new ListNode(0); // Dummy node to simplify logic
        ListNode* current = dummyHead;
        int carry = 0;

        while (l1 != nullptr || l2 != nullptr || carry != 0) {
            int sum = carry;
```

```
            if (l1 != nullptr) {
                sum += l1->val;
                l1 = l1->next;
            }
            if (l2 != nullptr) {
                sum += l2->val;
                l2 = l2->next;
            }
            carry = sum / 10; // Calculate carry for the next digit
            current->next = new ListNode(sum % 10);
            current = current->next;
        }

        return dummyHead->next; // Return the next node as the head of the result
    }
};
ListNode* createList(int arr[], int n) {
    if (n == 0) return nullptr;
    ListNode* head = new ListNode(arr[0]);
    ListNode* tail = head;
    for (int i = 1; i < n; i++) {
        tail->next = new ListNode(arr[i]);
        tail = tail->next;
    }
    return head;
}

void printList(ListNode* head) {
    while (head != nullptr) {
        cout << head->val;
        if (head->next) cout << " -> ";
```

```
        head = head->next;
    }
    cout << endl;
}

int main() {
    int arr1[] = {2, 4, 3};
    int arr2[] = {5, 6, 4};
    int n1 = sizeof(arr1) / sizeof(arr1[0]);
    int n2 = sizeof(arr2) / sizeof(arr2[0]);

    ListNode* l1 = createList(arr1, n1);
    ListNode* l2 = createList(arr2, n2);

    cout << "List 1: ";
    printList(l1);
    cout << "List 2: ";
    printList(l2);

    Solution sol;
    ListNode* result = sol.addTwoNumbers(l1, l2);

    cout << "Result: ";
    printList(result);
    return 0;
}
```

OUTPUT:




4)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".

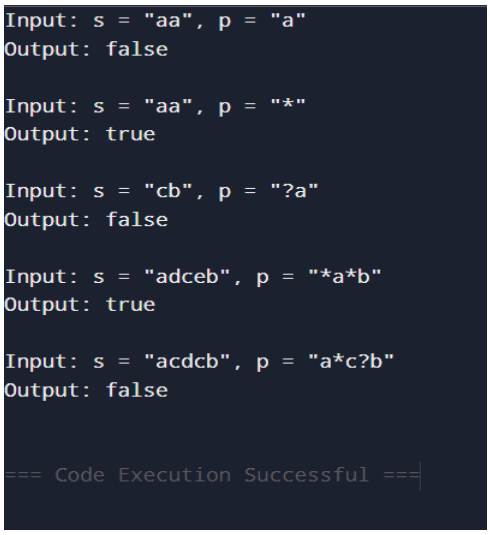
**SOLUTION:**

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

class Solution {
public:
    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;

        // Handle patterns with '*', as they can match an empty sequence
        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] == s[i - 1] || p[j - 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() {
    Solution sol;
    string s1 = "aa";
    string p1 = "a";
    cout << boolalpha << "Input: s = \"" << s1 << "\", p = \"" << p1 << "\"\n";
    cout << "Output: " << sol.isMatch(s1, p1) << "\n\n";
    string s2 = "aa";
    string p2 = "*";
```

```
    cout << "Input: s = \"" << s2 << "\", p = \"" << p2 << "\"\n";
    cout << "Output: " << sol.isMatch(s2, p2) << "\n\n";

    string s3 = "cb";
    string p3 = "?a";
    cout << "Input: s = \"" << s3 << "\", p = \"" << p3 << "\"\n";
    cout << "Output: " << sol.isMatch(s3, p3) << "\n\n";

    string s4 = "adceb";
    string p4 = "*a*b";
    cout << "Input: s = \"" << s4 << "\", p = \"" << p4 << "\"\n";
    cout << "Output: " << sol.isMatch(s4, p4) << "\n\n";

    string s5 = "acdcb";
    string p5 = "a*c?b";
    cout << "Input: s = \"" << s5 << "\", p = \"" << p5 << "\"\n";
    cout << "Output: " << sol.isMatch(s5, p5) << "\n";
    return 0;
}
```

**OUTPUT:**



5)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.

**SOLUTION:**

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

class Solution {
public:
    string makeLargestSpecial(string s) {
        vector<string> substrings;
        int count = 0, start = 0;

        // Split the string into special substrings
        for (int i = 0; i < s.size(); ++i) {
            count += (s[i] == '1') ? 1 : -1;
            if (count == 0) {
                // Recursively process the inner substring
                string inner = makeLargestSpecial(s.substr(start + 1, i - start - 1));
                substrings.push_back("1" + inner + "0");
                start = i + 1;
            }
```

```
        }

        // Sort substrings in reverse lexicographical order
        sort(substrings.rbegin(), substrings.rend());

        // Concatenate substrings
        string result = "";
        for (const string& sub : substrings) {
            result += sub;
        }

        return result;
    }
};

int main() {
    Solution sol;

    // Example 1
    string s1 = "11011000";
    cout << "Input: " << s1 << "\n";
    cout << "Output: " << sol.makeLargestSpecial(s1) << "\n\n";

    // Additional Test Cases
    string s2 = "101100";
    cout << "Input: " << s2 << "\n";
    cout << "Output: " << sol.makeLargestSpecial(s2) << "\n\n";

    string s3 = "111000";
    cout << "Input: " << s3 << "\n";
    cout << "Output: " << sol.makeLargestSpecial(s3) << "\n\n";
```

```
    return 0;
}
```

**OUTPUT:**