Day -2

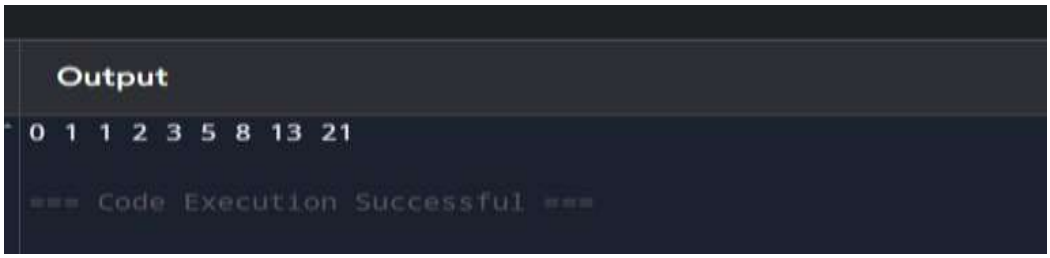
Student Name: sweta singh **UID -22BCS10664**
Branch: BE-CSE **Section/Group:** KPIT-901-A

1. Fibonacci Series Using Recursion

Code:

```
#include <stdio.h>
void printFib(int n) {
    if (n < 1) {
      printf("Invalid Number of terms\n");
      return;
   }
    int prev1 = 1;
   int prev2 = 0;
    for (int i = 1; i <= n; i++) {
        if (i > 2) {
          int curr = prev1 + prev2;
          prev2 = prev1;
          prev1 = curr;
          printf("%d ", curr);
       }
       else if (i == 1)
          printf("%d ", prev2);
       else if (i == 2)
          printf("%d ", prev1);
   }
}

int main() {
   int n = 9;
    printFib(n);
   return 0;
```

Output :



Q2. . **Reverse Linked List.**

Code: #include<stdio.h>

#include<stdlib.h>

struct Node

{

int data;

struct Node* next;

};

static void reverse(struct Node** head_ref)

{

struct Node* prev = NULL;

struct Node* current = *head_ref;

struct Node* next;

while (current != NULL)

{

next = current->next;

current->next = prev;

prev = current;

current = next;

}

*head_ref = prev;

}

```
void push(struct Node** head_ref, int new_data)
{struct Node* new_node =
                    (struct Node*) malloc(sizeof(struct Node));
          new_node->data = new_data;
          new_node->next = (*head_ref);
          (*head_ref) = new_node;
}
void printList(struct Node *head)
struct Node *temp = head;
     while(temp != NULL)
     {
          printf("%d ", temp->data);
          temp = temp->next;
     }
}
int main()
{
struct Node* head = NULL;
   push(&head, 1);
     push(&head, 2);
     push(&head, 3);
     push(&head, 4);

     printf("Given linked list\n");
     printList(head);
     reverse(&head);
     printf("\nReversed Linked list \n");
     printList(head);
     getchar();
```

Output :

Q3. **Add Two Numbers.**

Code: #include <iostream>

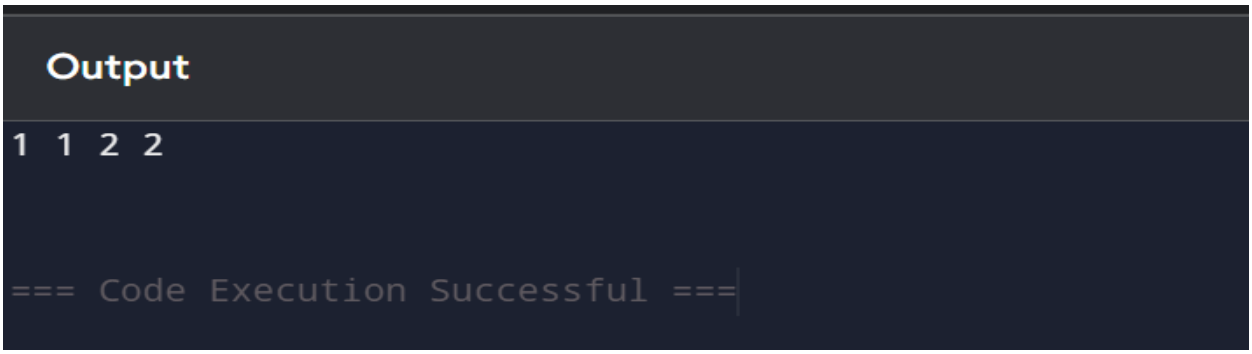
using namespace std;

```
class Node {
  public:
    int data;
    Node *next;
    Node(int val) {
        data = val;
        next = nullptr;
    }
};

Node *reverse(Node *head) {
    Node *prev = nullptr, *curr = head, *next = nullptr;

    while (curr != nullptr) {
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
    return prev;
}

Node *addTwoLists(Node *num1, Node *num2) {
```

```
    Node *res = nullptr, *curr = nullptr;
    int carry = 0;

    num1 = reverse(num1);
    num2 = reverse(num2);

    while (num1 != nullptr || num2 != nullptr || carry != 0) {
        int sum = carry;

        if (num1 != nullptr) {
            sum += num1->data;
            num1 = num1->next;
        }

        if (num2 != nullptr) {
            sum += num2->data;
            num2 = num2->next;
        }

        Node *newNode = new Node(sum % 10);
        carry = sum / 10;

        if (res == nullptr) {
            res = newNode;
            curr = newNode;
        } else {
            curr->next = newNode;
            curr = curr->next;
        }
    }

    return reverse(res);
}
```

```
void printList(Node *head) {
    Node *curr = head;
    while (curr != nullptr) {
        cout << curr->data << " ";
        curr = curr->next;
    }
    cout << "\n";
}

int main() {
    Node *num1 = new Node(1);
    num1->next = new Node(2);
    num1->next->next = new Node(3);

    Node *num2 = new Node(9);
    num2->next = new Node(9);
    num2->next->next = new Node(9);

    Node *sum = addTwoLists(num1, num2);
    printList(sum);

    return 0;
}
```

Output:



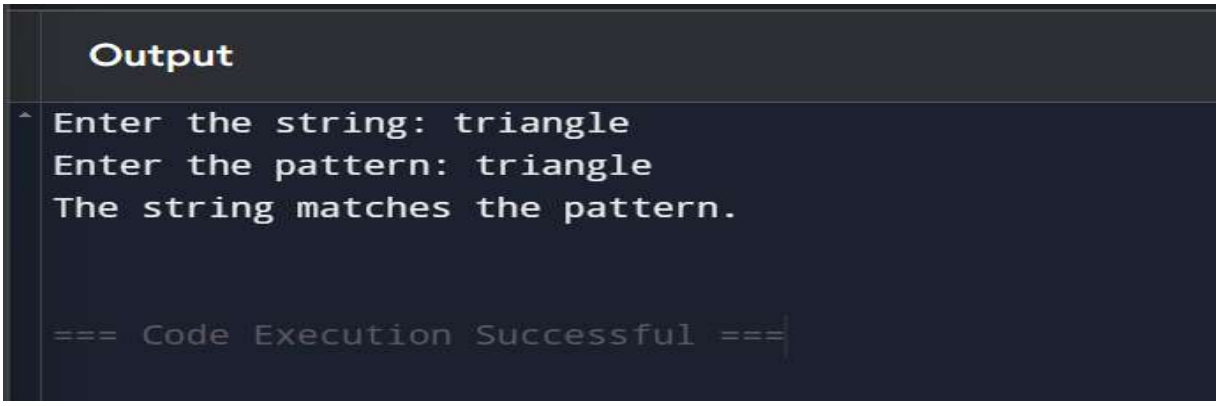
Q4. **Wildcard Matching**

Code:

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
bool isMatch(const char *str, const char *pattern) {
    int strLen = strlen(str);
    int patLen = strlen(pattern);
    bool dp[strLen + 1][patLen + 1];
    memset(dp, false, sizeof(dp));
    dp[0][0] = true;
   for (int j = 1; j <= patLen; j++) {
        if (pattern[j - 1] == '*') {
            dp[0][j] = dp[0][j - 1];
        }
    }
    for (int i = 1; i <= strLen; i++) {
        for (int j = 1; j <= patLen; j++) {
            if (pattern[j - 1] == '*') {

                dp[i][j] = dp[i][j - 1] || dp[i - 1][j];
            } else if (pattern[j - 1] == '?' || pattern[j - 1] == str[i - 1]) {

                dp[i][j] = dp[i - 1][j - 1];
            } else {
                dp[i][j] = false;
            }
        }
    }
   return dp[strLen][patLen];
}int main() {
    char str[100], pattern[100];
   printf("Enter the string: ");
    scanf("%s", str);
```

```
    printf("Enter the pattern: ");
    scanf("%s", pattern);
    if (isMatch(str, pattern)) {
        printf("The string matches the pattern.\n");
    } else {
        printf("The string does not match the pattern.\n");
    }
return 0;
}\
```

Output:



Q5. **. Special Binary String.**

Code: #include <stdio.h>

#include <stdlib.h>

#include <string.h>

```
// Comparator for sorting strings in descending order
int compareDesc(const void *a, const void *b) {
    return strcmp(*(const char **)b, *(const char **)a);
}

// Helper function to process a special binary string
void processSpecialBinaryString(char *s, char *result, int start, int end) {
    if (start >= end) {
        return;
    }
```

```
char **substrings = (char **)malloc((end - start) * sizeof(char *));
int count = 0, balance = 0, last = start;
int substrCount = 0;

// Split into valid substrings
for (int i = start; i < end; i++) {
    if (s[i] == '1') {
        balance++;
    } else {
        balance--;
    }

    // A valid special substring is found
    if (balance == 0) {
        substrings[substrCount] = (char *)malloc((i - last + 2) * sizeof(char));
        strncpy(substrings[substrCount], &s[last], i - last + 1);
        substrings[substrCount][i - last + 1] = '\0';

        // Process the substring recursively
        processSpecialBinaryString(substrings[substrCount], substrings[substrCount], 1, i - last);
        substrCount++;
        last = i + 1;
    }
}

// Sort substrings in descending order
qsort(substrings, substrCount, sizeof(char *), compareDesc);

// Construct the result string
for (int i = 0; i < substrCount; i++) {
    strcat(result, substrings[i]);
    free(substrings[i]);
}
```

```
    free(substrings);
}

char *makeLargestSpecial(char *s) {
    int len = strlen(s);
    char *result = (char *)malloc((len + 1) * sizeof(char));
    result[0] = '\0';

    processSpecialBinaryString(s, result, 0, len);

    return result;
}

int main() {
    char s[100];

    printf("Enter the special binary string: ");
    scanf("%s", s);

    char *result = makeLargestSpecial(s);
    printf("Largest lexicographical special binary string: %s\n", result);

    free(result);
    return 0;
}
```

Output: