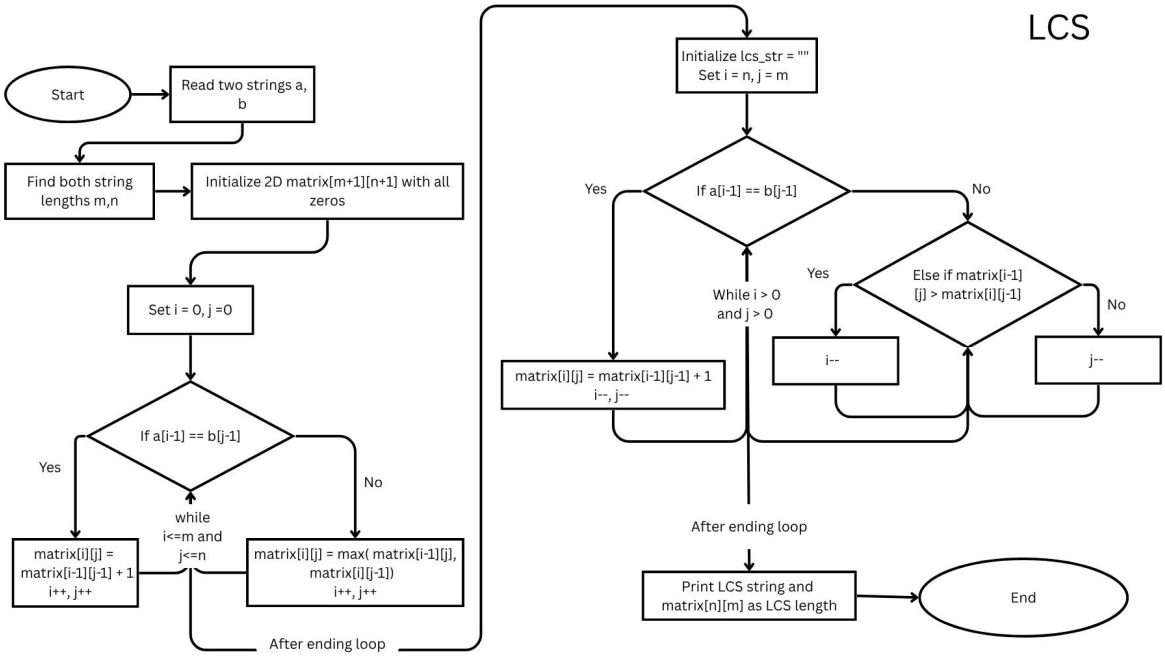
Problem Statement: Stepwise Execution Analysis of LCS

Theory:

Algorithm:

1. Make a table to store results for every pair of prefixes of the two strings.
2. Set the first row and first column to zero since an empty string has no common subsequence.
3. Compare characters of both strings one by one and fill the table.
4. If the characters match, add 1 to the diagonal value; if not, take the larger of the top or left cell.
5. The value in the bottom-right cell of the table is the length of the longest common subsequence.

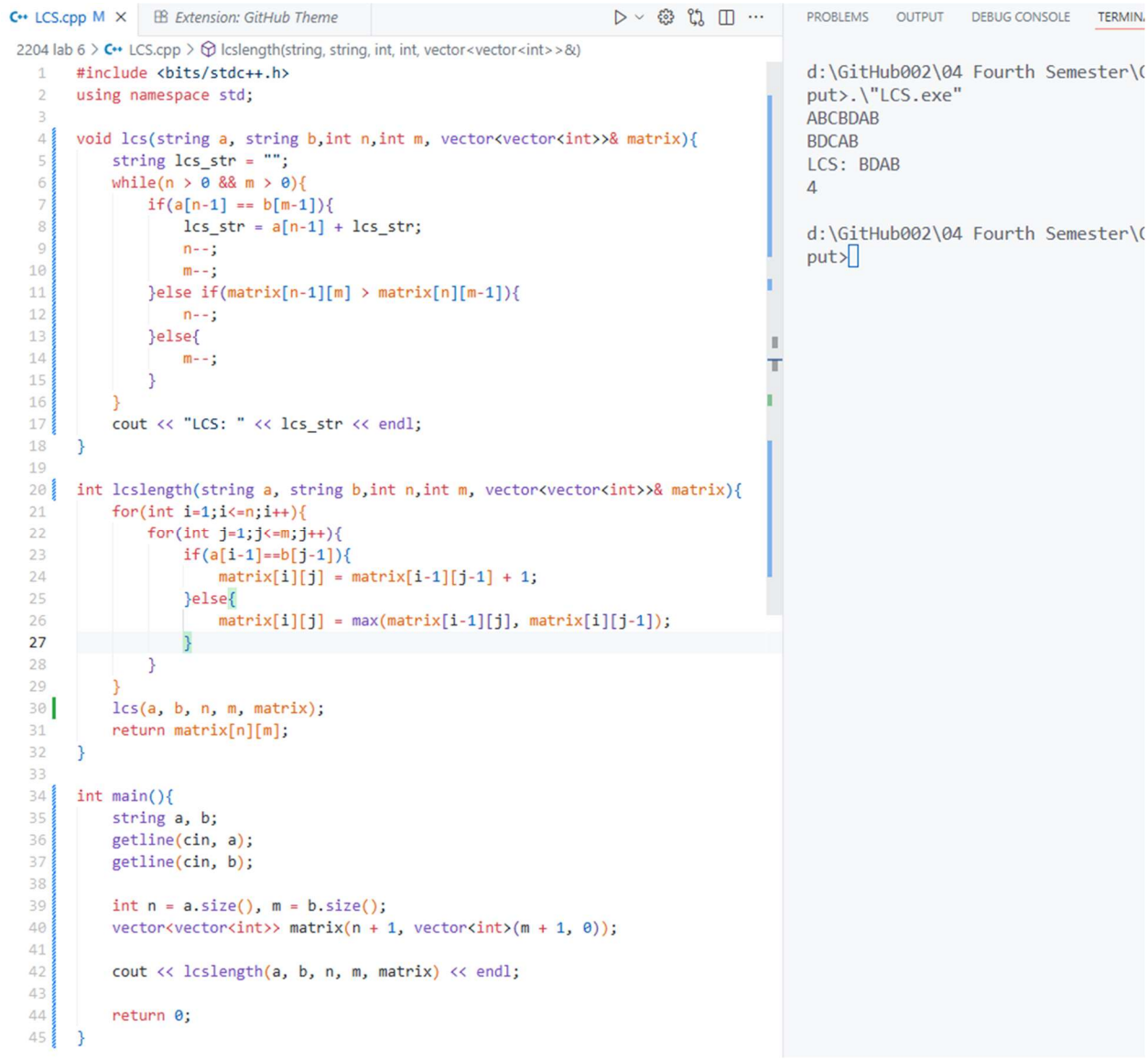
Flowchart:



Code:

```
#include <bits stdc++.h="">
using namespace std;

void lcs(string a, string b,int n,int m, vector<vector<int>>& matrix){
    string lcs_str = "";
    while(n > 0 && m > 0){
        if(a[n-1] == b[m-1]){
            lcs_str = a[n-1] + lcs_str;
            n--;
            m--;
        }else if(matrix[n-1][m] > matrix[n][m-1]){
            n--;
        }else{
            m--;
        }
    }
    cout << "LCS: " << lcs_str << endl;
}

int lcslength(string a, string b,int n,int m, vector<vector<int>>& matrix){
    for(int i=1;i<=n;i++){
        for(int j=1;j<=m;j++){
            if(a[i-1]==b[j-1]){
                matrix[i][j] = matrix[i-1][j-1] + 1;
            }else{
                matrix[i][j] = max(matrix[i-1][j], matrix[i][j-1]);
            }
        }
    }
    lcs(a, b, n, m, matrix);
    return matrix[n][m];
}

int main(){
    string a, b;
    getline(cin, a);
    getline(cin, b);

    int n = a.size(), m = b.size();
    vector<vector<int>> matrix(n + 1, vector<int>(m + 1, 0));

    cout << lcslength(a, b, n, m, matrix) << endl;

    return 0;
}
```

Screenshots:



Analysis:

Initialize string

A = "ABCBDAB"

B = "BDCAB"

Lengths: m = 7, n = 5

We'll build a `(m+1) × (n+1)` matrix (8×6).

Row 0 and Column 0 are initialized to 0.

Initial Matrix (Before Any Pass)

idx	0	1	2	3	4	5	6
0		**0**	**B**	**D**	**C**	**A**	**B**
1	**0**	0	0	0	0	0	0
2	**A**	0	0	0	0	0	0
3	**B**	0	0	0	0	0	0
4	**C**	0	0	0	0	0	0
5	**B**	0	0	0	0	0	0
6	**D**	0	0	0	0	0	0
7	**A**	0	0	0	0	0	0
8	**B**	0	0	0	0	0	0

- Pass 1 → i = 1 (A vs BDCAB)

idx	0	1	2	3	4	5	6
0			**B**	**D**	**C**	**A**	**B**
1	**0**	0	0	0	0	0	0
2	**A**	0	0	0	0	1	1
3	**B**	0	0	0	0	0	0
4	**C**	0	0	0	0	0	0
5	**B**	0	0	0	0	0	0
6	**D**	0	0	0	0	0	0
7	**A**	0	0	0	0	0	0
8	**B**	0	0	0	0	0	0

Found match at A–A (position 1,4)

- Pass 2 → i = 2 (B vs BDCAB)

idx	0	1	2	3	4	5	6
0			**B**	**D**	**C**	**A**	**B**
1	**0**	0	0	0	0	0	0
2	**A**	0	0	0	0	1	1
3	**B**	0	1	1	1	1	2
4	**C**	0	0	0	0	0	0
5	**B**	0	0	0	0	0	0
6	**D**	0	0	0	0	0	0
7	**A**	0	0	0	0	0	0
8	**B**	0	0	0	0	0	0

Matches: B–B (1,1) and B–B (2,5)

- Pass 3 → i = 3 (C vs BDCAB)

idx	0	1	2	3	4	5	6
0			**B**	**D**	**C**	**A**	**B**
1	**0**	0	0	0	0	0	0
2	**A**	0	0	0	0	1	1
3	**B**	0	1	1	1	1	2
4	**C**	0	1	1	2	2	2
5	**B**	0	0	0	0	0	0
6	**D**	0	0	0	0	0	0
7	**A**	0	0	0	0	0	0
8	**B**	0	0	0	0	0	0

Match: C–C (3,3).

- Pass 4 → i = 4 (B vs BDCAB)

idx	0	1	2	3	4	5	6
0			**B**	**D**	**C**	**A**	**B**
1	**0**	0	0	0	0	0	0
2	**A**	0	0	0	0	1	1
3	**B**	0	1	1	1	1	2
4	**C**	0	1	1	2	2	2
5	**B**	0	1	1	2	2	3
6	**D**	0	0	0	0	0	0
7	**A**	0	0	0	0	0	0
8	**B**	0	0	0	0	0	0

Match: B–B (4,1) and B–B (4,5)

- Pass 5 → i = 5 (D vs BDCAB)

idx	0	1	2	3	4	5	6
0			**B**	**D**	**C**	**A**	**B**
1	**0**	0	0	0	0	0	0
2	**A**	0	0	0	0	1	1
3	**B**	0	1	1	1	1	2
4	**C**	0	1	1	2	2	2
5	**B**	0	1	1	2	2	3
6	**D**	0	1	2	2	2	3
7	**A**	0	0	0	0	0	0
8	**B**	0	0	0	0	0	0

Match: D–D (5,2)

- Pass 6 → i = 6 (A vs BDCAB)

idx	0	1	2	3	4	5	6
0			**B**	**D**	**C**	**A**	**B**
1	**0**	0	0	0	0	0	0
2	**A**	0	0	0	0	1	1
3	**B**	0	1	1	1	1	2
4	**C**	0	1	1	2	2	2
5	**B**	0	1	1	2	2	3
6	**D**	0	1	2	2	2	3
7	**A**	0	1	2	2	3	3
8	**B**	0	0	0	0	0	0

Match: A–A (6,4)

- Pass 7 → i = 7 (B vs BDCAB)

idx	0	1	2	3	4	5	6
0			**B**	**D**	**C**	**A**	**B**
1	**0**	0	0	0	0	0	0
2	**A**	0	0	0	0	1	1
3	**B**	0	1	1	1	1	2
4	**C**	0	1	1	2	2	2
5	**B**	0	1	1	2	2	3
6	**D**	0	1	2	2	2	3
7	**A**	0	1	2	2	3	3
8	**B**	0	1	2	2	3	4

Matches: B–B (7,1) and B–B (7,5)

And this is the Final Matrix.

Result -> LCS Length: 4

Traceback (to find the LCS string)

- Start from bottom-right (7,5):
- A[6]=B, B[4]=B → add 'B'
- Move diagonally (6,4)
- A[5]=A, B[3]=A → add 'A'
- Move diagonally (5,3)
- A[4]=D, B[2]=C → move up
- A[3]=B, B[2]=C → move left
- A[2]=C, B[2]=C → add 'C'
- Move diagonally (2,1)
- A[1]=B, B[0]=B → add 'B'

idx	0	1	2	3	4	5	6
0			**B**	**D**	**C**	**A**	**B**
1	**0**	0	0	0	0	0	0
2	**A**	0	0	0	0	1	1
3	**B**	0	1	1	1	1	2
4	**C**	0	1	1	2	2	2
5	**B**	0	1	1	2	2	3
6	**D**	0	1	2	2	2	3
7	**A**	0	1	2	2	3	3
8	**B**	0	1	2	2	3	4

Reverse order → LCS = "BCA

Conclusion:

The Longest Common Subsequence (LCS) algorithm efficiently identifies the longest sequence of characters that appear in the same order in two strings, though not necessarily consecutively. Using Dynamic Programming, it breaks the problem into smaller subproblems and stores their results in a matrix to avoid redundant calculations. This makes it significantly faster than a simple recursive approach. The LCS algorithm is especially useful in applications like DNA sequence analysis, text comparison, and version control systems. However, due to its O(n × m) time and space complexity, it becomes less efficient for very large strings. For small to medium-sized datasets, LCS provides an optimal balance between accuracy and performance, making it a reliable choice for sequence comparison tasks.