K. J. Somaiya College of Engineering, Mumbai-77
(A Constituent College of Somaiya Vidyavihar University)
Department of Computer Engineering

Batch: A1	**Roll No.: 16010123012**
Experiment No. 10	
Grade: AA / AB / BB / BC / CC / CD /DD	
Signature of the Staff In-charge with date	

Title: Study, Implementation, and Analysis of the Longest Common Subsequence Algorithm.

Objective: To compute longest common subsequence for the given two strings.

CO to be achieved:

CO 2	Analyze and solve problems for divide and conquer strategy, greedy method, dynamic programming approach and backtracking and branch & bound policies.
CO 3	Analyze and solve problems for different string matching algorithms.

Books/ Journals/ Websites referred:

1. **Ellis horowitz, Sarataj Sahni, S.Rajsekaran," Fundamentals of computer algorithm", University Press**
2. **T.H.Cormen ,C.E.Leiserson,R.L.Rivest and C.Stein," Introduction to algortihtms",2nd Edition ,MIT press/McGraw Hill,2001**
3. **http://www.math.utah.edu/~alfeld/queens/queens.**

Pre Lab/ Prior Concepts:
Data structures, Concepts of algorithm analysis

Historical Profile:
Given 2 sequences, $X = x1 , ..., xm$ and $Y = y1 , ... , yn$, find a subsequence common to both whose length is longest. A subsequence doesn't have to be consecutive, but it has to be in order.

New Concepts to be learned:
String matching algorithm, Dynamic programming approach for LCS, Applications of LCS.

Recursive **Formulation:**

Define $c[i, j]$ = length of LCS of X_i and Y_j.
Final answer will be computed with $c[m, n]$.

c[i, j]= 0
if i=0 or j=0.
c[i, j]= c[i − 1, j − 1] + 1
if i,j>0 and xi=yj

c[i, j]= max(c[i − 1, j], c[i, j − 1])
if i, j > 0 and x_i <> y_j

Algorithm: Longest Common Subsequence

Compute length of optimal solution-

LCS-LENGTH *(X , Y, m, n)*

for $i \leftarrow 1$ **to** m

do $c[i, 0] \leftarrow 0$

for $j \leftarrow 0$ **to** n

do $c[0, j] \leftarrow 0$

for $i \leftarrow 1$ **to** m

do for $j \leftarrow 1$ **to** n

do if $x_i = y_j$

then $c[i, j] \leftarrow c[i-1, j-1] + 1$

$b[i, j] \leftarrow$ "≈"

else if $c[i-1, j] \geq c[i, j-1]$

then $c[i, j] \leftarrow c[i-1, j]$

$b[i, j] \leftarrow$ "↑"

else $c[i, j] \leftarrow c[i, j-1]$

$b[i, j] \leftarrow$ "←"

return c and b

Print the solution-

PRINT-LCS*(b, X , i, j)*

if $i = 0$ or $j = 0$

then return if

$b[i, j]$ = "≈"

then PRINT-LCS$(b, X, i-1, j-1)$

print *xi*

elseif $b[i, j]$ = "↑"

then PRINT-LCS$(b, X, i - 1, j)$

else PRINT-LCS$(b, X, i, j - 1)$

Initial call is PRINT-LCS(b, X, m, n).

$b[i, j]$ points to table entry whose subproblem we used in solving LCS of X_i and Y_j.

When $b[i, j] = \approx$, we have extended LCS by one character. So longest com- mon subsequence = entries with ≈ in them.

Example: LCS computation/ Analysis of LCS computation:

LCS

$$L[i,j] = \begin{cases} 0 & ; \text{if } i=0 \text{ or } j=0 \\ L[i-1][j-1]+1 & ; \text{if } i,j>0 \text{ \& } x_i = y_j \\ \text{move}(L[i-1][j], L[i][j-1]) & ; \text{if } i,j>0 \text{ \& } x_i \neq y_j \end{cases}$$

x = notebook
Y = facebook

	y_j	f	a	c	e	b	o	o	k
x_i	0	0	0	0	0	0	0	0	0
n	0	0↑	0↑	0↑	0↑	0↑	0↑	0↑	0↑
o	0	0↑	0↑	0↑	0↑	0↑	1↖	1↖	1←
t	0	0↑	0↑	0↑	0↑	0↑	1↑	1↑	1↑
e	0	0↑	0↑	0↑	1↖	1←	1↑	1↑	1↑
b	0	0↑	0↑	0↑	1↑	2↖	2←	2←	2←
o	0	0↑	0↑	0↑	1↑	2↑	3↖	3←	3←
o	0	0↑	0↑	0↑	1↑	2↑	3↑	4↖	4←
k	0	0↑	0↑	0↑	1↑	2↑	3↑	4↑	5↖

LCS length : 5
LCS : ebook

Implementation:

```cpp
#include <bits/stdc++.h>
using namespace std;

void printDPMatrix(const vector<vector<int>> &dp, const string &X, const string &Y)
{
  cout << "\nDP Matrix (LCS lengths):\n";
  cout << "     ";
  for (int j = 0; j < Y.length(); j++)
  {
    cout << Y[j] << " ";
  }
  cout << endl;

  for (int i = 0; i <= X.length(); i++)
  {
    if (i == 0)
    {
      cout << "  ";
    }
    else
    {
      cout << X[i - 1] << " ";
    }

    for (int j = 0; j <= Y.length(); j++)
    {
      cout << dp[i][j] << " ";
    }
    cout << endl;
  }
  cout << endl;
}

pair<string, vector<vector<int>>> lcs(const string &X, const string &Y)
{
  int m = X.length();
  int n = Y.length();

  vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
  vector<vector<char>> direction(m + 1, vector<char>(n + 1));
  for (int i = 1; i <= m; ++i)
  {
    for (int j = 1; j <= n; ++j)
    {
      if (X[i - 1] == Y[j - 1])
```

```
      {
        dp[i][j] = dp[i - 1][j - 1] + 1;
        direction[i][j] = 'D';
      }
      else if (dp[i - 1][j] >= dp[i][j - 1])
      {
        dp[i][j] = dp[i - 1][j];
        direction[i][j] = 'U';
      }
      else
      {
        dp[i][j] = dp[i][j - 1];
        direction[i][j] = 'L';
      }
    }
  }

  string lcs_str;
  lcs_str.reserve(dp[m][n]);

  int i = m, j = n;
  while (i > 0 && j > 0)
  {
    if (direction[i][j] == 'D')
    {
      lcs_str.push_back(X[i - 1]);
      i--;
      j--;
    }
    else if (direction[i][j] == 'U')
    {
      i--;
    }
    else
    {
      j--;
    }
  }

  reverse(lcs_str.begin(), lcs_str.end());
  return {lcs_str, dp};
}

int main()
{
  string X, Y;

  cout << "Enter the first string: ";
```

```
  cin >> X;
  cout << "Enter the second string: ";
  cin >> Y;

  auto [result, dp] = lcs(X, Y);

  cout << "LCS of \"" << X << "\" and \"" << Y << "\" is \"" << result << "\"" << endl;

  printDPMatrix(dp, X, Y);
}
```

Output:

```
PS D:\KJSCE\BTech\SY\Sem IV\AOA\Code> cd "d:\KJSCE\BTech\SY\Sem IV\AOA\Code\" ;
if ($?) { g++ lcs.cpp -o lcs } ; if ($?) { .\lcs }
Enter the first string: facebook
Enter the second string: notebook
LCS of "facebook" and "notebook" is "ebook"

DP Matrix (LCS lengths):
    n o t e b o o k
  0 0 0 0 0 0 0 0 0
f 0 0 0 0 0 0 0 0 0
a 0 0 0 0 0 0 0 0 0
c 0 0 0 0 0 0 0 0 0
e 0 0 0 0 1 1 1 1 1
b 0 0 0 0 1 2 2 2 2
o 0 0 1 1 1 2 3 3 3
o 0 0 1 1 1 2 3 4 4
k 0 0 1 1 1 2 3 4 5
```

Analysis of LCS computation:

Time Complexity: O(m × n), where m and n are the lengths of the two input strings
Space Complexity: O(m × n), where m and n are the lengths of the two input strings

Conclusion:

I have successfully completed the experiment on the Longest Common Subsequence (LCS) using the dynamic programming approach. Through the implementation, I learned how to break down the problem into overlapping subproblems and store intermediate results to improve efficiency.