**Open Ended Problems**

**Question 1**

**Aim-** Given two sequences, a subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. For example, “abc”, “abg”, “bdf”, “aeg”, „”acefg”, .. etc are subsequences of “abcdefg”. So a string of length n has 2^n different possible subsequences. It is a classic computer science problem, the basis of file comparison programs and has applications in bioinformatics. Develop a pro gram to implement the solution of Longest Common Sub-sequence problem.

**Theory-**

The Longest Common Subsequence (LCS) problem involves finding the longest subsequence common to two given sequences. A subsequence is a sequence derived from another sequence by deleting some or no elements, without changing the order of the remaining elements. LCS is a classic problem with many applications in fields like file comparison, version control systems, and bioinformatics (for DNA sequence alignment).

For example, given two strings "abcde" and "ace", the LCS would be "ace" because it is the longest subsequence that appears in both strings in the same relative order.

**Software Used –** Visual Studio Code

**Code-**

#include <iostream>

#include <vector>

#include <string>

using namespace std;

// Function to calculate LCS using memoization (efficient version)

int lcsMem(string &str1, string &str2, int n, int m, vector<vector<int> >& dp) {

if (n == 0 || m == 0) {

return 0; // Base case: one of the strings is empty

}

if (dp[n][m] != -1) {

return dp[n][m]; // Return already computed value (memoization)

}

if (str1[n - 1] == str2[m - 1]) {

// If last characters match, consider it and move both indices back

dp[n][m] = 1 + lcsMem(str1, str2, n - 1, m - 1, dp);

} else {

// If last characters don't match, either ignore one character from one of the strings

int ans1 = lcsMem(str1, str2, n - 1, m, dp);

int ans2 = lcsMem(str1, str2, n, m - 1, dp);

dp[n][m] = max(ans1, ans2); // Take the max of ignoring either character

}

return dp[n][m];

}

int main() {

// Take input from user

string str1, str2;

cout << "Enter the first sequence: ";

cin >> str1

cout << "Enter the second sequence: ";

cin >> str2

int n = str1.length();

int m = str2.length();

// DP table initialized to -1 for memoization

vector<vector<int> > dp(n + 1, vector<int>(m + 1, -1));

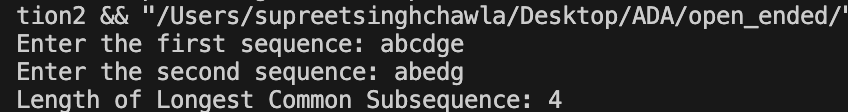
// Call the LCS function with indices for both strings

cout << "Length of Longest Common Subsequence: " << lcsMem(str1, str2, n, m, dp) << endl;

return 0;

}

**Output-**



**Question 2**

**Aim-** A road network can be considered as a graph with positive weights. The nodes represent road junctions and each edge of the graph is associated with a road segment between two junctions. The weight of an edge may correspond to the length of the associated road segment, the time needed to traverse the segment or the cost of traversing the segment. Using directed edges it is also possible to model one-way streets. Such graphs are special in the sense that some edges are more important than others for long distance travel (e.g. highways). This property has been formalized using the notion of highway dimension. There are a great number of algorithms that exploit this property and are therefore able to compute the shortest path a lot quicker than would be possible on general graphs. Develop a program to find the shortest path from each node to solve the road network problem.

**Theory-**

The Shortest Path Problem is a fundamental problem in graph theory and is widely used in road network analysis. A road network is modeled as a graph, where the nodes represent road junctions, and edges represent road segments, with weights corresponding to distances, times, or costs associated with traveling between junctions. The goal is to compute the shortest path from a given starting node (source) to all other nodes.

In many real-world scenarios, road networks exhibit hierarchical properties where certain roads (like highways) are more important for long-distance travel than others. Algorithms that exploit these properties, such as Dijkstra's Algorithm , can compute the shortest paths efficiently by prioritizing important roads.

**Software Used –** Visual Studio Code

**Code-**

#include <iostream>

#include <vector>

#include <queue>

#include <climits>

using namespace std;

class Edge {

public:

int v;

int wt;

Edge(int v, int wt) {

this->v = v;

this->wt = wt;

}

};

// Dijkstra's algorithm to find the shortest path from a given source vertex

void dijkstra(int src, vector<vector<Edge> > &graph, int V) {

// Min-heap priority queue (distance, vertex)

priority\_queue<pair<int, int>, vector<pair<int, int> >, greater<pair<int, int> > > pq;

vector<int> dist(V, INT\_MAX); // Distance from source to each vertex

dist[src] = 0;

pq.push(make\_pair(0, src)); // Push starting vertex with distance 0

while (!pq.empty()) {

int u = pq.top().second; // Get the vertex with the smallest distance

pq.pop();

// Explore neighbors of u

for (Edge e : graph[u]) {

int v = e.v;

int weight = e.wt;

// Relaxation step

if (dist[v] > dist[u] + weight) {

dist[v] = dist[u] + weight;

pq.push(make\_pair(dist[v], v));

}

}

}

// Print shortest distances from the source vertex

cout << "From Node " << src << ":\n";

cout << "Vertex Distance from Source" << endl;

for (int i = 0; i < V; i++) {

cout << i << " \t\t " << dist[i] << endl;

}

cout << endl;

}

int main() {

int V = 6; // Number of vertices

vector<vector<Edge> > graph(V);

// Add edges to the graph (directed graph)

graph[0].push\_back(Edge(1, 2));

graph[0].push\_back(Edge(2, 4));

graph[1].push\_back(Edge(2, 1));

graph[1].push\_back(Edge(3, 7));

graph[2].push\_back(Edge(4, 3));

graph[3].push\_back(Edge(5, 1));

graph[4].push\_back(Edge(3, 2));

graph[4].push\_back(Edge(2, 5));

// Run Dijkstra's algorithm from each vertex

for (int i = 0; i < V; i++) {

dijkstra(i, graph, V);

}

return 0;

}

**Output-**

