Floyd Warshall Algorithm

The Floyd–Warshall algorithm works by maintaining a two-dimensional array that represents the distances between nodes. Initially, this array is filled using only the direct edges between nodes. Then, the algorithm gradually updates these distances by checking if shorter paths exist through intermediate nodes.

This algorithm works for both the directed and undirected weighted graphs and can handle graphs with both positive and negative weight edges.

Note: It does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative).

Idea Behind Floyd Warshall Algorithm:
Suppose we have a graph dist[][] with V vertices from 0 to V-1. Now we have to evaluate a dist[][] where dist[i][j] represents the shortest path between vertex i to j.

Let us assume that vertices i to j have intermediate nodes. The idea behind Floyd Warshall algorithm is to treat each and every vertex k from 0 to V-1 as an intermediate node one by one. When we consider the vertex k, we must have considered vertices from 0 to k-1 already. So we use the shortest paths built by previous vertices to build shorter paths with vertex k included.

Step-by-step implementation

Start by updating the distance matrix by treating each vertex as a possible intermediate node between all pairs of vertices.
Iterate through each vertex, one at a time. For each selected vertex k, attempt to improve the shortest paths that pass through it.
When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices.
For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.
k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.
k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j], if dist[i][j] > dist[i][k] + dist[k][j]
Repeat this process for each vertex k until all intermediate possibilities have been considered.

Longest Common Subsequence (LCS)

Given two strings, s1 and s2, the task is to find the length of the Longest Common Subsequence. If there is no common subsequence, return 0. A subsequence is a string generated from the original string by deleting 0 or more characters, without changing the relative order of the remaining characters.

For example, subsequences of “ABC” are “”, “A”, “B”, “C”, “AB”, “AC”, “BC” and “ABC”. In general, a string of length n has 2n subsequences.

The idea is to compare the last characters of s1 and s2. While comparing the strings s1 and s2 two cases arise:

Match : Make the recursion call for the remaining strings (strings of lengths m-1 and n-1) and add 1 to result.
Do not Match : Make two recursive calls. First for lengths m-1 and n, and second for m and n-1. Take the maximum of two results.
Base case : If any of the strings become empty, we return 0.

[Expected Approach 1] Using Bottom-Up DP (Tabulation) – O(m * n) Time and O(m * n) Space

[Expected Approach 2] Using Bottom-Up DP (Space-Optimization):