☐ Problem No: 03

☐ **Problem name:** Suppose that the relation R_1 and R_2 on a set A are represented by the matrices

$M_{R1} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$ and $M_{R2} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix}$. Write a program to find the $M_{R1 \cup R2}$ and $M_{R1 \oplus R2}$.

Theory: The program shown above defines two function. 'matrix_union()' and 'matrix_exclusive_or()', to compute the union and symmetric union (exclusive or) of two matrices respectively.

The matrix_union() function takes two matrics as input and returns a new matrics that represented the union of two input matrics. It uses list comperihension to perform element-wise logical OR operation between the element of the two input matrics.

CamScanner

'The matrix_exclusive_or()' function also takes two matrics as input and return a new matrics that represent the symmetric difference of the two input matrics. It uses list comperihension and the X-OR operator (^) to perform element-wise logical X-OR operation between the elements of the two input matrics.

After defining this functions, the program creates two matrics R1 and R2 which represent relations. These matric are then passed to the matric_union() and matrix_exclusive_or() functions to compute the union and symmetric difference $MR1 \oplus R2$.

Here a python program is written below:

```
def matrix_union(matrix1, matrix2):
    union_result = [[matrix1[i][j] or matrix2[i][j]
            for j in range(len(matrix1[0]))]
            for i in rang(len(matrix1))]
                return union_result
```

Function to compute the matrix Union (element-wise OR) of the two matrix.

```
def matrix_~~de~~exclusive_OR (matrix1, matrix2):
XOR_result = [[matrix1 [i] [j] ^ matrix2[i] [j]
for j in range (len (matrix1 [0]))] for i in range (len (matrix1))]
return XOR_result.
```

Function to compute the matrix exclusive_OR (elementwise XOR) of the two matrix.

```
R1= [[1, 0, 1],     # Give the matrix representing relation
     [1, 0, 0],       R1 and R2.
     [0, 1, 1]]

R2= [[1, 0, 1]
     [0, 1, 1]
     [1, 0, 1]]
```

~~MR1⊕~~

```
mR1_xor_R2 = matrix_exclusive_or (R1, R2) # find mR1⊕R2
mR1_union_ R2= matrix_union(R1, R2)        and MR1 U R2.
print(" First Matrix = ", R1)
print(" Second Matrix= ", R2)
print ("Rows = ", len(R1), "Cols = ", len (R1[0]))
print (" Matrix union ~~RR~~ MR1UMR2", mR1_union_R2)
```

print(" Matrix Exclusive OR MR1 ⊕ R2", MR1 ⊕ R2)
# print the result,

Output:

First Matrix = [[1, 0, 1], [1, 0, 0], [0, 1, 1]]
Second matrix = [[1, 0, 1], [0, 1, 1], [1, 0, 1]]
Rows = 3 Cols = 3
Matrix Union MR1 ∪ R2: [[1, 0, 1], (1, 1, 1], [1, 1, 1]]
Matrix Exclusive OR MR1 ⊕ R2: [[0, 0, 0], (1, 1, 1], (1, 1, 0]]

4

Problem no- 4

Problem name: Write a program to find shortest path by Warshall's algorithm.

Theory: The program shown above applies the Floyd-Warshall algorithm to find the shortest paths in a graph. The graph is represented by an adjacency matrix.

The function 'floydWarshall()' takes two algorith: the number of vertices in the graph and the adjacency matrix representing the graph. within this function, the floyd warshall algorithm is implimented using nested forloops

After applying the floyd warshall algorithm, the program prints the resulting adjacency matrix, showing the distance between each pair of vertices. The leftmost row represents the origin vertex and the topmost column represents the destination vertex. The values in the matrix represent the shortest distance between the corresponding vertex.

In the main() function the number of vertices is set to 4. The adjancency matrix is defined ~~best~~ based on the given graph.

Finally the 'main()' function calls the 'floydWarshall()' function with the number of vertics and the adjacency matrix as arguments to find the shortest paths in the graph.

Here a C++ program is written given below:

```
#include<iostream>
using namespace std;          // Defining the infinity value.
const int INF = 1000000000;

// Function to apply Floyd-Warshall algorithm
void floydWarshall(int vertex, int adjancency_matrix[][4])
{
  // Iterate over all vertex as intermediate nodes
  for (int k=0: k< vertex; k++)
  {
   // for each pair of vertex (i,j), check if going to vertex
      k provides a short path
   for (int i = 0; i< vertex; i++)
```

P.T.O.

```
{ for( int j=0; j< vertex ; j++)
{// Relax the distance from i to j by allowing vertex k as an
intermediate vertex
// considers which one is better, going through vertex k, or
the previous value
adjacency_matrix[i][j]=min( adjancency_matrix[i][j],
        adjacency_matrix [i][k] + adjacency_matrix [k][j]);
}
}
}

// Pretty print the graph
// o/d means the leftmost row is the origin vertex.
// and the topmost column as distination vertex

cout<<" o/d" ;
for (int i=0; i< vertex; i++)
{ cout<<'\t'<< i+1;
}
cout<< endl;
```

```
// printing the adjacency matrix
for(int i=0; i<vertex; i++)
{ cout<<i+1;
for(int j=0;j<vertex; j++)
{ cout<<'t'<<adjacency_matrix[i][j];
}
cout<<endl;
}}
int main()
{ // Namber of vertex in the graph
int vertex=4;

/*
Input is given as adjacency matrix,
input represent this undirected graph

A--1--B
|    /
3   /
|  1
| /
C--2--D
should set infinite value for each pair of vertex
that has no edge */
```

```
// The adjacency matrix represented the graph
int adjacency_matrix [4][4] = {
{0, 5, INF, 10},
{INF, 0, 3, INF},
{INF, INF, 0, 1},
{INF, INF, INF, 0}};

// find the shortest path using Floyd-Warshall algorithm
by call the function

floydWarshall(vertex, adjacency_matrix);
return 0;
}
```

Output:

0/d	1	2	3	4
1	0	5	8	9
2	1000000000	0	3	4
3	1000000000	1000000000	0	1
4	1000000000	1000000000	1000000000	0