Discipline Core

ALGORITHMS

CS4003

Assignment 3

Sheikh Muhammed Tadeeb (AU19B1014)

❖ Problem Statement:

➢ Description:

Mr. ABC decided to take his family for a Ashtavinayak Yatra from Pune, he found that there are total 8 temples located in the range of 100 KMs around Pune in different directions and it takes one full day to visit all the temples and return back to Pune in the night. The details of all the paths are as follows:

Distance of Ashtavinayak temples from Pune

On Pune - Solapur road
Pune - Theur (Shree Chintamani) 25 Kms
Pune - Siddhatek (Shree Siddhivinayak) 98 Kms
Pune - Moregaon (Shree Moreshwar) 64 Kms

On Pune - Nagar road
Pune - Ranjangaon (Shree Mahaganapati) 50 Kms

On Pune - Nashik road
Pune - Ozar (Shree Vighnahar) 85 Kms
Pune - Lenyandri (Shree Girijatmaj) 94 Kms

On Pune - Mumbai road
Pune - Mahad (Shree Varadavinayak) 83 Kms
Pune - Pali (Shree Ballaleshwar) 110 Kms

➢ Instructions:

Mr. ABC is new to Pune and want to find the best suitable path so that all his journey can get completed in one day. You are being his friend and he seeks some help from you to get the shortest route to travel to all the cities. You must implement the suitable algorithm to find the shortest path from Pune to all other cities and help Mr. ABC in completing his journey via the shortest path.

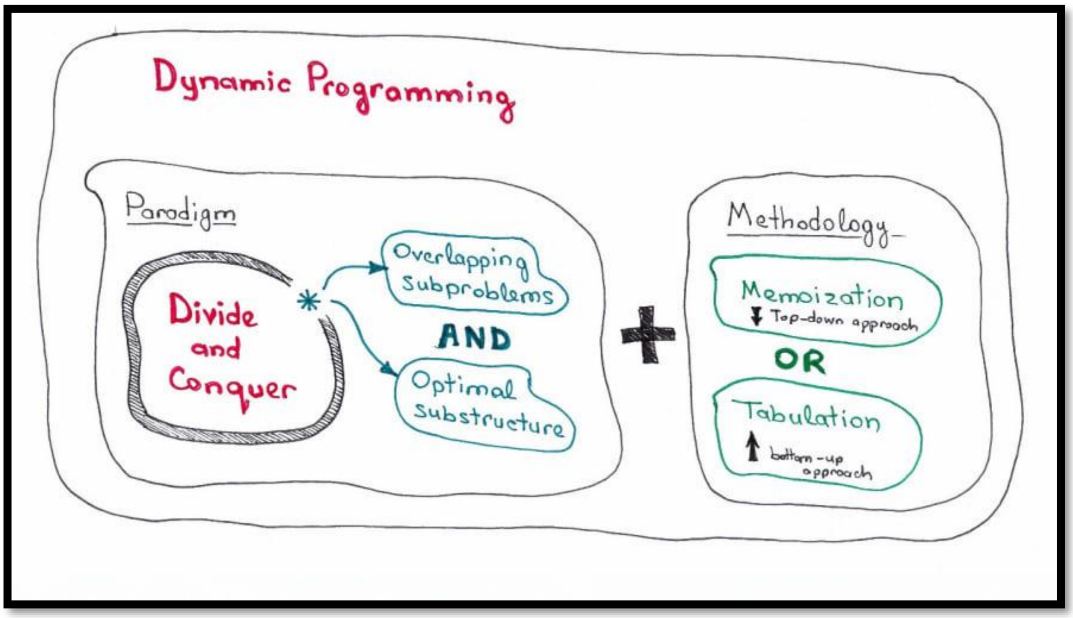
❖ Theory:

➢ **Dynamic Programming:**

Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of sub problems, so that we do not have to re-compute them when needed later. The two important factors of dynamic programming are: -

1. Optimal substructure:- Optimal solution can be constructed from optimal solutions of its sub problems

2. Overlapping sub-problems:- Problem can be broken down into sub problems which are reused several times or a recursive algorithm for the problem solves the same sub problems.



❖ C++ Code:

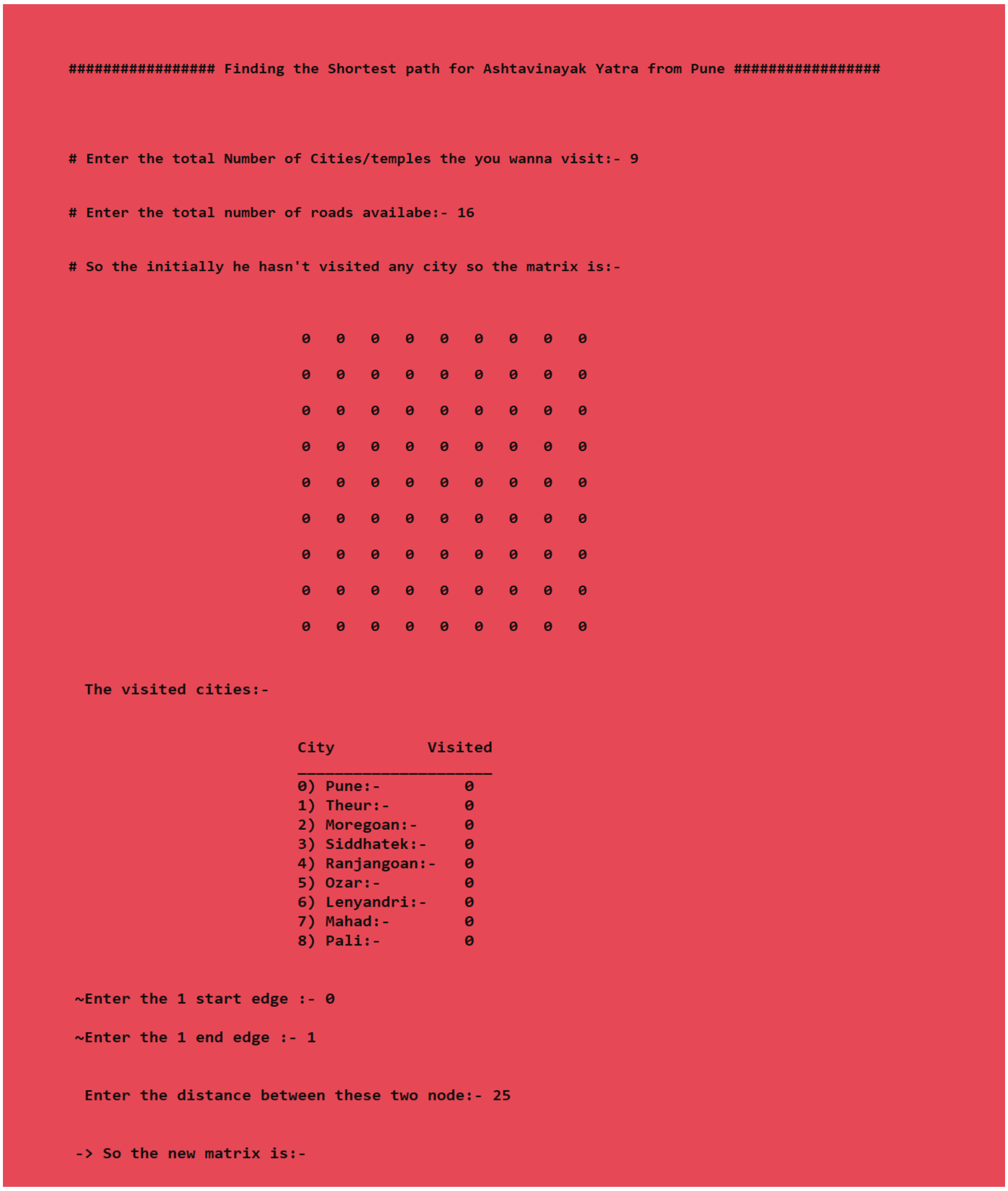
```cpp
#include <bits/stdc++.h>
using namespace std;

// Parent class.
class Graph
{
        // Defining member variables.
            // Number of vertex.
            int v;
            // Number of edges.
            int e;
            // Adjacency matrix.
            int** adj;
        public:
               // Defining member functions.
            // To create the initial adjacency matrix.
            void graph(int v,int visited[]);
            // Function to insert a new edge.
            void addEdge(int start, int e);
            // Function to display the Minimum weight hamilton
cycle/traversal.
            void best_route(int start,int visited[],int vertex);
            // Function to find the least number in a row.
        int least(int c,int visited[],int vertex);
        //
        void visited_mark(int visited[],int v);
          // Distructor of this class.
          ~Graph(){};
};

// Initial total distance is considered as 0.
int cost = 0;

/* Single Inheritance and
this class is the child class*/
class Calling : protected Graph
{
     // Defining member variables.
          int v,startedge, endedge,e,source;
     public:
          // Defining constructor for automatic initialisation.
          Calling(){
          // Formatting
          cout<<endl<<endl<<endl<<endl<<endl<<"\t################# Finding the Shortest path for Ashtavinayak Yatra from Pune #################";
             // Taking total number of nodes.
               cout<<endl<<endl<<endl<<endl<<endl<<"\t# Enter the total Number of Cities/temples the you wanna visit:- ";
             cin>>v;
         // MAking an array to check which nodes had been visited.
               int visited[v];
              // Taking total number of edges.
             cout<<endl<<endl<<"\t# Enter the total number of roads availabe:- ";
             cin>>e;
             // Creating the initial graph
             cout<<endl<<endl<<"\t# So the initially he hasn't visited any city so the matrix is:- "<<endl;
             // Creating the object of above class.
               Graph G;
               // Calling graph function of above class.
             G.graph(v,visited);
             // Taking the routes from different connections
             for (int i = 0; i < e; i++){
                  // Taking user inputs.
                    cout<<endl<<endl<<"\t~Enter the "<<i+1<<" start edge :- ";  // 0 to 1
                  cin>>startedge;
                  cout<<endl<<"\t~Enter the "<<i+1<<" end edge :- ";
                  cin>>endedge;
```

```
                        // Calling a function to create a matrix for this connection.
                        G.addEdge(startedge,endedge);
                }
                // User Controlled system.
                    int n;
                    int i = 0;
                    cout<<endl<<endl<<endl<<"\t# For how many different starting point you want to check:- ";
                    cin>>n;
                // Running for different nos of times.
                        while(i<n){
                            cout<<endl<<endl<<endl<<"\tEnter the source node or starting point of Pilgrims:- ";
                                cin>>source;
                                cout<<endl<<"\tThe best/shortest route is:- ";
                                G.best_route(source,visited,v);  // initial point 0
                                cout<<endl<<endl<<endl<<"\t\t Minimum total distance is:- "<<cost<<" Kms"<<endl;
                                i++;
                            }
                    }
};

// Function to fill the empty adjacency matrix
void Graph :: graph(int v,int visited[])
{
       // Putting Total nos of nodes/vertex
    this->v = v;
    // Creating a pointer array of type int and storing its 0 index address in adj pointer.
    adj = new int* [v];
    // Creating matrix.
    for (int row = 0; row < v; row++) {
        /* At [row] index of adj address we are
              creating further arrays and storing their values at [row] index. */
              adj[row] = new int[v];
        // Now filling the columns of further arrays as 0 , initially.
              for (int column = 0; column < v; column++) {
             adj[row][column] = 0;
               }
               visited[row] = 0;
           }
           // Formatting
           cout<<endl<<endl;
           // Used to display the created Graph Matrix.
    for (int row = 0; row < v; row++){
       cout<<endl<<"\t\t\t\t";
        for (int column = 0; column < v; column++) {
            cout<<"   "<<adj[row][column];
        }
        cout<<endl;
    }
    visited_mark(visited,v);
}


void Graph :: visited_mark(int visited[],int v){
       // Formatting.
    cout<<endl<<endl<<"\t The visited cities:- "<<endl;
    cout<<endl<<endl<<"\t\t\t\tCity          Visited"<<endl;
    cout<<"\t\t\t\t_____________________"<<endl;
    // Used to display the visited nodes.
       for (int row = 0; row < v; row++){
       cout<<"\t\t\t\t";
        if(row == 0){
             cout<<"0) Pune:-         "<<visited[row];
             }else if(row == 1){
                    cout<<"1) Theur:-        "<<visited[row];
             }else if(row == 2){
                    cout<<"2) Moregoan:-     "<<visited[row];
             }else if(row == 3){
                    cout<<"3) Siddhatek:-    "<<visited[row];
             }else if(row == 4){
                    cout<<"4) Ranjangoan:-   "<<visited[row];
             }else if(row == 5){
                    cout<<"5) Ozar:-         "<<visited[row];
             }else if(row == 6){
                    cout<<"6) Lenyandri:-    "<<visited[row];
             }else if(row == 7){
                    cout<<"7) Mahad:-        "<<visited[row];
             }else if(row == 8){
                    cout<<"8) Pali:-         "<<visited[row];
             }
        cout<<endl;
    }
}




// Function to add an edge to the graph.
void Graph::addEdge(int start, int end)
{
```

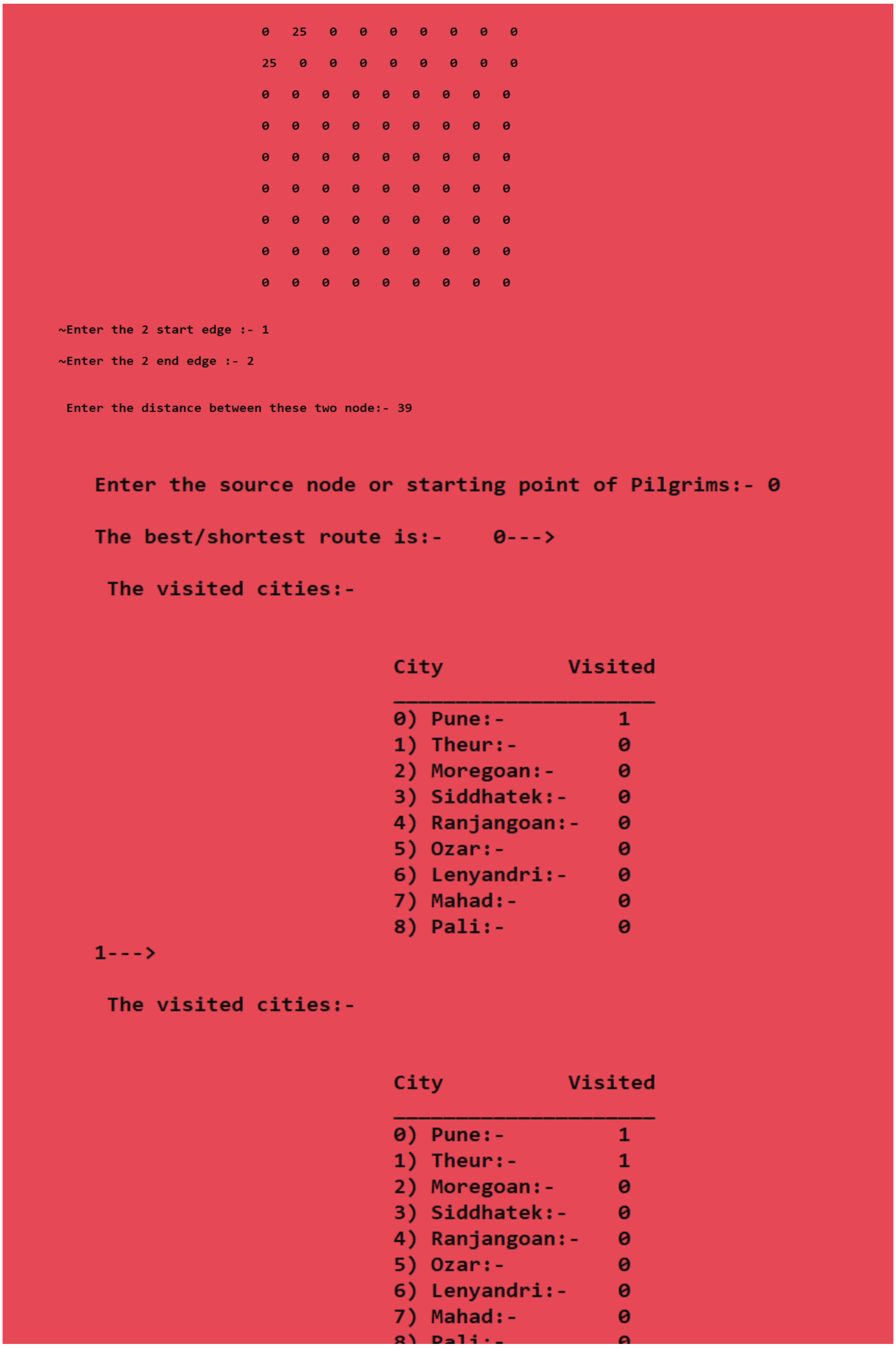
```cpp
        // Considering a bidirectional edge.
        int dist;
        // Taking distances between two nodes.
          cout<<endl<<endl<<"\t Enter the distance between these two node:- ";
        cin>>dist;
        // Creating an Adjacency matrix for the same.
        adj[start][end] = dist;
        adj[end][start] = dist;
        // Displaying the newly created adjacency matrix.
        cout<<endl<<endl<<"\t-> So the new matrix is:- "<<endl<<endl;
        for (int row = 0; row < v; row++){
          cout<<endl<<"\t\t\t\t";
            for (int column = 0; column < v; column++) {
                cout<<"   "<<adj[row][column];
            }
          cout<<endl;
          }

    }

    // Finding the least available distance from each point.
    int Graph :: least(int c,int visited[],int vertex)
    {
        // Initialising the variables.
          int i,nc=999;
          int min=999,kmin;

          for(i=0;i<vertex;i++)
          {
                  /* Checking if the node is visited or not and if
                  their exist a connecting road between them or not
                  and checking the min in the row */
                  if( (adj[c][i]!=0) && (visited[i]==0) && (adj[c][i]< min) ){

                              min = adj[c][i];
                              // storing index of min in nc.
                              nc = i;
                  }

          }
        // Used for calculating total distance.
          if(min != 999){
                  cost += min;
        }
          // Returning the index of minimum.
          return nc;
```

```
}

/* Function to find minimum weight
hamilton cycle. */
void Graph :: best_route(int city,int visited[],int vertex)
{
        // Varibale declaration.
        int i,ncity;
     // Updating the value of visited city in array.
        visited[city] = 1;
     // Formatting
        cout<<"\t"<<city<<"--->";
        // Finding the next least distance from this point.
        ncity = least(city,visited,vertex);
        // Formatting
        visited_mark(visited,v);
     // Base condition.
        if(ncity==999)
        {
                ncity=0;
                cout<<ncity;
                cost += adj[city][ncity];
                return;
        }
        // Reccursion
        best_route(ncity,visited,vertex);
}

// Driver code
int main()
{
        // Used for changing Screen background.
        system("Color C0");
     // Declaring object of child class.
        Calling obj;
     return 0;
}

```

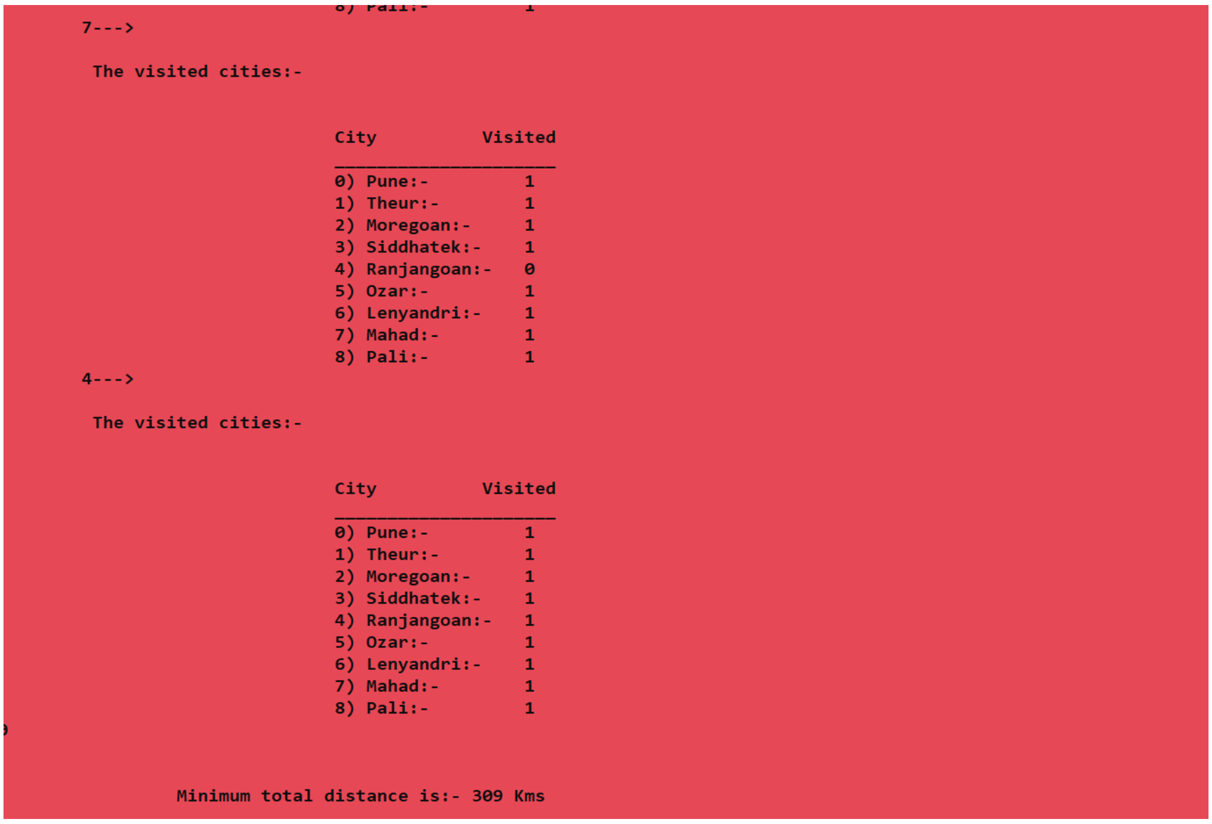
❖ C++ Output:



```
################# Finding the Shortest path for Ashtavinayak Yatra from Pune #################


# Enter the total Number of Cities/temples the you wanna visit:- 9

# Enter the total number of roads availabe:- 16

# So the initially he hasn't visited any city so the matrix is:-


                           0   0   0   0   0   0   0   0   0

                           0   0   0   0   0   0   0   0   0

                           0   0   0   0   0   0   0   0   0

                           0   0   0   0   0   0   0   0   0

                           0   0   0   0   0   0   0   0   0

                           0   0   0   0   0   0   0   0   0

                           0   0   0   0   0   0   0   0   0

                           0   0   0   0   0   0   0   0   0

                           0   0   0   0   0   0   0   0   0


  The visited cities:-


                          City          Visited
                          ______________________
                          0) Pune:-        0
                          1) Theur:-       0
                          2) Moregoan:-    0
                          3) Siddhatek:-   0
                          4) Ranjangoan:-  0
                          5) Ozar:-        0
                          6) Lenyandri:-   0
                          7) Mahad:-       0
                          8) Pali:-        0


 ~Enter the 1 start edge :- 0

 ~Enter the 1 end edge :- 1


  Enter the distance between these two node:- 25


 -> So the new matrix is:-
```



```
                              0   25   0   0   0   0   0   0   0

                              25   0   0   0   0   0   0   0   0

                              0   0   0   0   0   0   0   0   0

                              0   0   0   0   0   0   0   0   0

                              0   0   0   0   0   0   0   0   0

                              0   0   0   0   0   0   0   0   0

                              0   0   0   0   0   0   0   0   0

                              0   0   0   0   0   0   0   0   0

                              0   0   0   0   0   0   0   0   0


 ~Enter the 2 start edge :- 1

 ~Enter the 2 end edge :- 2

  Enter the distance between these two node:- 39


    Enter the source node or starting point of Pilgrims:- 0

    The best/shortest route is:-    0--->

     The visited cities:-


                                        City          Visited
                                        _____________________
                                        0) Pune:-           1
                                        1) Theur:-          0
                                        2) Moregoan:-       0
                                        3) Siddhatek:-      0
                                        4) Ranjangoan:-     0
                                        5) Ozar:-           0
                                        6) Lenyandri:-      0
                                        7) Mahad:-          0
                                        8) Pali:-           0
    1--->

     The visited cities:-


                                        City          Visited
                                        _____________________
                                        0) Pune:-           1
                                        1) Theur:-          1
                                        2) Moregoan:-       0
                                        3) Siddhatek:-      0
                                        4) Ranjangoan:-     0
                                        5) Ozar:-           0
                                        6) Lenyandri:-      0
                                        7) Mahad:-          0
                                        8) Pali:-           0
```



```
                                8) Pali:-          1
       7--->

        The visited cities:-


                                City          Visited
                                _____________________
                                0) Pune:-          1
                                1) Theur:-         1
                                2) Moregoan:-      1
                                3) Siddhatek:-     1
                                4) Ranjangoan:-    0
                                5) Ozar:-          1
                                6) Lenyandri:-     1
                                7) Mahad:-         1
                                8) Pali:-          1
       4--->

        The visited cities:-


                                City          Visited
                                _____________________
                                0) Pune:-          1
                                1) Theur:-         1
                                2) Moregoan:-      1
                                3) Siddhatek:-     1
                                4) Ranjangoan:-    1
                                5) Ozar:-          1
                                6) Lenyandri:-     1
                                7) Mahad:-         1
                                8) Pali:-          1



                 Minimum total distance is:- 309 Kms
```

❖ Time Complexity:

Since we are solving this using Dynamic Programming, we know that Dynamic Programming approach contains sub-problems.

Here after reaching i^{th} node finding remaining minimum distance to that i^{th} node is a sub-problem.

If we solve recursive equation we will get total $(n-1)\ 2^{(n-2)}$ sub problems, which is $O\ (n2^n)$.

Each sub-problem will take $O\ (n)$ time (finding path to remaining (n-1) nodes).

Therefore, total time complexity is $O\ (n2^n) * O\ (n) = O\ (n^2 2^n)$

❖ Conclusion:

Dynamic programming is when you use solutions to smaller sub problems in order to solve a larger problem. This is easiest to implement recursively because you usually think of such solutions in terms of a recursive function. An iterative implementation is usually preferred though, because it takes less time and memory.

Memoization is used to prevent recursive DP implementation from taking a lot more time than needed. Most times, a DP algorithm will use the same sub problem in solving multiple large problems. In a recursive implementation, this means we will re-compute the same thing multiple times. Memoization implies saving the results of these sub problems into a table. When entering a recursive call, we check if its result exists in the table: if yes, we return it, if not, we compute it, save it in the table, and then return it.