M.Sc. (Five Year Integrated) in Computer Science
(Artificial Intelligence & Data Science)

Third Semester

Laboratory Record

21-805-0306: ALGORITHMS LAB

Submitted in partial fulfillment
of the requirements for the award of degree in
Master of Science (Five Year Integrated)
in Computer Science (Artificial Intelligence & Data Science) of
Cochin University of Science and Technology (CUSAT)
Kochi

Submitted by

AKSHADHA A
(80521004)

DEPARTMENT OF COMPUTER SCIENCE
COCHIN UNIVERSITY OF SCIENCE AND TECHNOLOGY (CUSAT)
KOCHI-682022

JANUARY 2023

DEPARTMENT OF COMPUTER SCIENCE

COCHIN UNIVERSITY OF SCIENCE AND TECHNOLOGY (CUSAT)

KOCHI, KERALA-682022

*This is to certify that the software laboratory record for **21-805-0306: Algorithms Lab** is a record of work carried out by **AKSHADHA A(80521004)**, in partial fulfillment of the requirements for the award of degree in **Master of Science (Five Year Integrated)** in **Computer Science (Artificial Intelligence & Data Science)** of Cochin University of Science and Technology (CUSAT), Kochi. The lab record has been approved as it satisfies the academic requirements in respect of the third semester laboratory prescribed for the Master of Science (Five Year Integrated) in Computer Science degree.*

Faculty Member in-charge

Mrs.Raheena Salihin
Guest Faculty
Department of Computer Science
CUSAT

Dr. Philip Samuel
Professor and Head
Department of Computer Science
CUSAT

Table of Contents

QUICK SORT

AIM

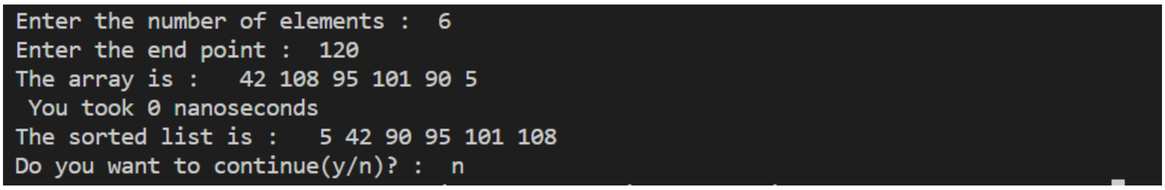
To sort the elements using quick sort and to determine the time required to sort the elements.

PROGRAM

```
#include<iostream>
#include <ctime>
#include <iomanip>
#include<cstdlib>
#include<chrono>
using namespace std;
using namespace std::chrono;
int Partition(int *A,int LB,int UB)
{
    int pivot = A[LB];
    int START = LB;
    int END = UB;
    while(START < END)
    {
        while(A[START] <= pivot)
        {
            START++;
        }
        while(A[END] > pivot)
        {
            END--;
        }
        if(START < END)
        {
            int temp = A[START];
            A[START] = A[END];
            A[END] = temp;
        }
    }
    int t1 = A[LB];
    A[LB] = A[END];
    A[END] = t1;
    return END;
}
```

```
void QuickSort(int *A,int LB,int UB)
{
    if (LB < UB)
    {
        int LOC = Partition(A,LB,UB);
        QuickSort(A,LB,LOC-1);
        QuickSort(A,LOC+1,UB);
    }
}
void display(int *A, int n)
{
    cout<<"The sorted list is : "<<"  ";
    for(int i = 0; i<n; i++)
    {
        cout<<A[i]<<" ";
    }

}
int main()
{
    int n;
    char choice;
    do
    {
        cout<<"Enter the number of elements : "<<" ";
        cin>>n;
        int A[n];
        int endpt;
        cout<<"Enter the end point : "<<" ";
        cin>>endpt;
        for(int i = 0; i<n; i++)
        {
            A[i] = 1+rand()%endpt;
        }
        cout<<"The array is : "<<"  ";
        for(int i = 0; i<n; i++)
        {
            cout<<A[i]<<" ";
        }
        cout<<endl;
        int LB = 0;
        int UB = n;
        auto start = high_resolution_clock::now();
        QuickSort(A,LB,UB);
        auto stop = high_resolution_clock::now();
        auto doneTime = duration_cast<microseconds>(stop-start);
        cout<< " You took " <<doneTime.count() << " nanoseconds\n";
        display(A,n);
        cout<<endl;
        cout<<"Do you want to continue(y/n)? : "<<" ";
        cin>>choice;
    } while (choice!='n');
}
```

SAMPLE INPUT-OUTPUT



```
Enter the number of elements :  6
Enter the end point :  120
The array is :   42 108 95 101 90 5
 You took 0 nanoseconds
The sorted list is :   5 42 90 95 101 108
Do you want to continue(y/n)? :  n
```

BREADTH FIRST SEARCH

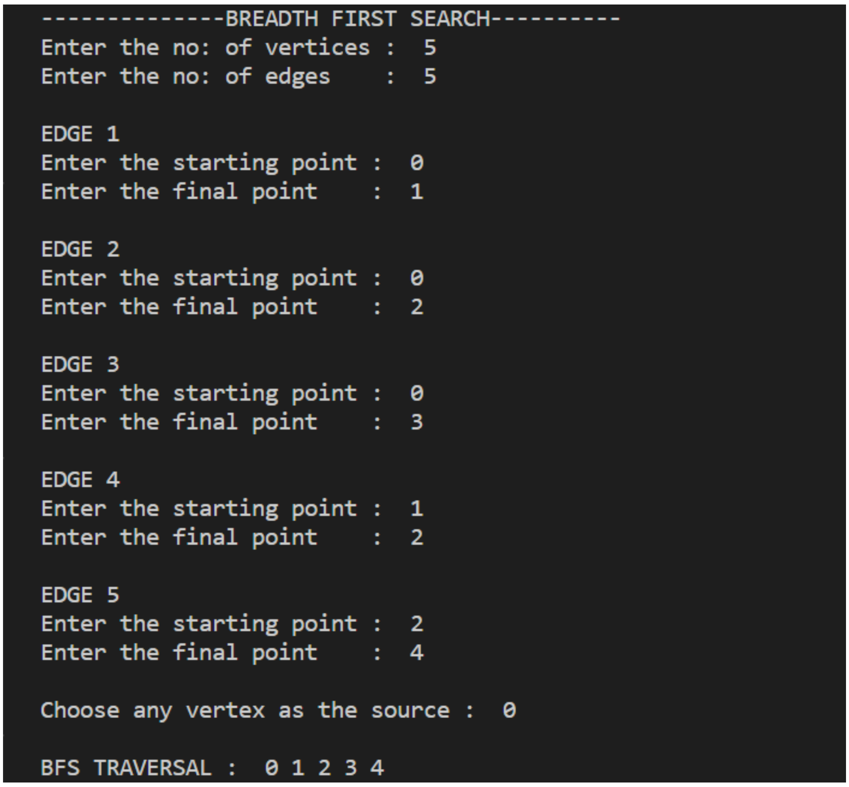
AIM

To print all the nodes reachable from a given starting node in a digraph using BFS method

PROGRAM

```
#include<iostream>
#include<vector>
#include<queue>
using namespace std;
void add_edge(vector<int>adj[],int u,int v)
{
    adj[u].push_back(v);
}
void bfs(int source,vector<int>adj[],bool visited[])
{
    queue<int>q;
    q.push(source);
    visited[source] = true;
    while(!q.empty())
    {
        int u = q.front();
        cout<<u<<" ";
        q.pop();
        //Traversal
        for(int i = 0;i<adj[u].size();i++)
        {
            if(!visited[adj[u][i]])
            {
                q.push(adj[u][i]);
                visited[adj[u][i]] = true;
            }
        }
    }
}
int main()
{
    cout<<"--------------BREADTH FIRST SEARCH----------"<<endl;
    int n,e;
    cout<<"Enter the no: of vertices : "<<" ";
    cin>>n;
    vector<int>adj[n];
    bool visited[n];
    for(int i = 0; i<5; i++)
    {
        visited[i] = false;
    }
    cout<<"Enter the no: of edges    : "<<" ";
    cin>>e;
    int a,b,s;
    for(int i = 0 ; i<e;i++)
    {
        cout<<endl;
        cout<<"EDGE "<<i+1<<endl;
        cout<<"Enter the starting point : "<<" ";
        cin>>a;
        cout<<"Enter the final point    : "<<" ";
        cin>>b;
        add_edge(adj,a,b);
    }
    cout<<endl;
    cout<<"Choose any vertex as the source : "<<" ";
    cin>>s;
    cout<<endl;
    cout<<"BFS TRAVERSAL : "<<" ";
    bfs(s,adj,visited);
    cout<<endl;
}
```

SAMPLE INPUT-OUTPUT



```
--------------BREADTH FIRST SEARCH----------
Enter the no: of vertices :  5
Enter the no: of edges    :  5

EDGE 1
Enter the starting point :  0
Enter the final point    :  1

EDGE 2
Enter the starting point :  0
Enter the final point    :  2

EDGE 3
Enter the starting point :  0
Enter the final point    :  3

EDGE 4
Enter the starting point :  1
Enter the final point    :  2

EDGE 5
Enter the starting point :  2
Enter the final point    :  4

Choose any vertex as the source :  0

BFS TRAVERSAL :  0 1 2 3 4
```

DEPTH FIRST SEARCH

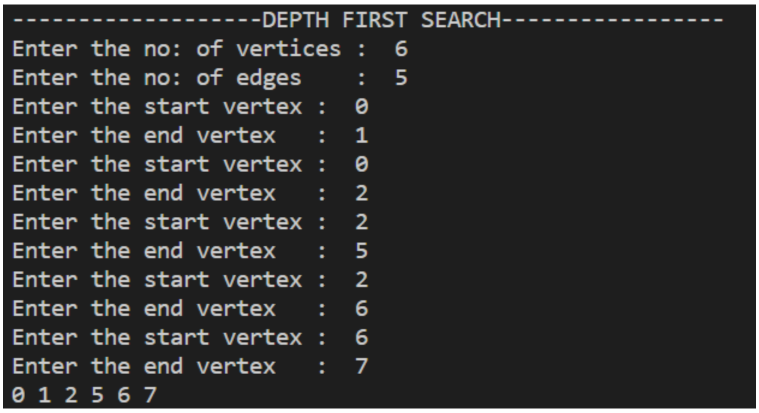
AIM

To check whether a given graph is connected using DFS method

PROGRAM

```
#include <iostream>
#include <vector>
using namespace std;

vector<int> g[10];
bool visited[10] = { false };

void create_edge(int src, int dest) {
    g[src].push_back(dest);
}

void dfs(int vertex) {
    if (visited[vertex] == true) return;
    cout << vertex << " ";

    visited[vertex] = true;
    for (auto node: g[vertex]) {
        dfs(node);
    }
}

int main() {
    int n,e,u,v;
    cout<<"-------------------DEPTH FIRST SEARCH-----------------"<<endl;
    cout<<"Enter the no: of vertices : "<<" ";
    cin>>n;
    cout<<"Enter the no: of edges    : "<<" ";
    cin>>e;
    for(int i = 0 ; i < e ; i++)
    {
        cout<<"Enter the start vertex : "<<" ";
        cin>>u;
        cout<<"Enter the end vertex   : "<<" ";
        cin>>v;
        create_edge(u, v);
    }
    dfs(0);

}
```

SAMPLE INPUT-OUTPUT



```
-------------------DEPTH FIRST SEARCH-----------------
Enter the no: of vertices :  6
Enter the no: of edges    :  5
Enter the start vertex :  0
Enter the end vertex   :  1
Enter the start vertex :  0
Enter the end vertex   :  2
Enter the start vertex :  2
Enter the end vertex   :  5
Enter the start vertex :  2
Enter the end vertex   :  6
Enter the start vertex :  6
Enter the end vertex   :  7
0 1 2 5 6 7
```

DIJKSTRA'S ALGORITHM

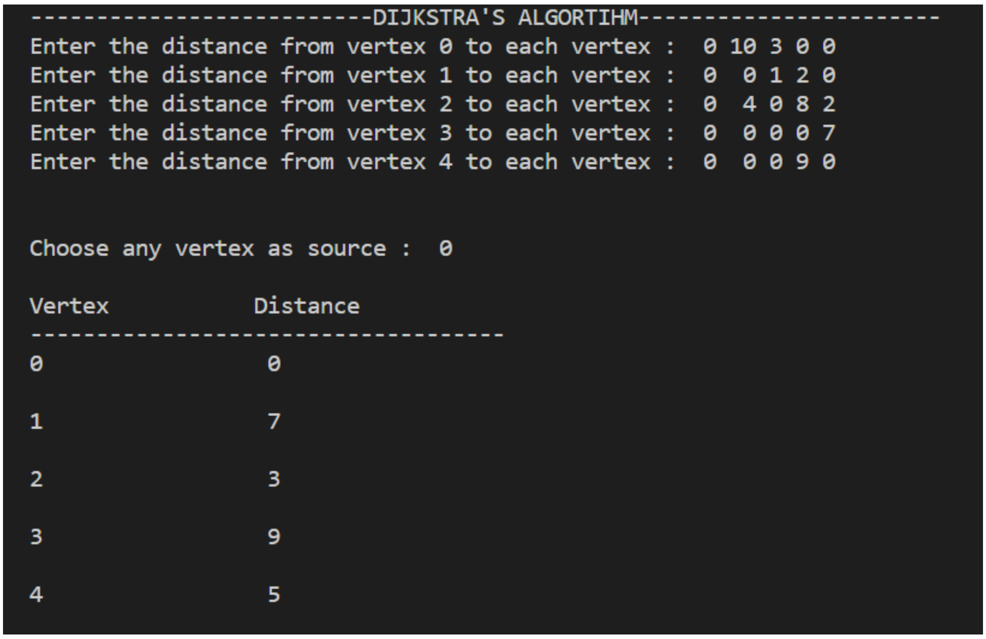
AIM

To find shortest paths to other vertices from a given vertex in a weighted connected graph using Dijkstra's algorithm

PROGRAM

```
#include<iostream>
#include<stdio.h>
using namespace std;
#define INF 9999
#define V 5

void dijkstra(int G[V][V],int num,int start)
{
    int cost[V][V];
    int distance[V],pred[V];
    int visited[V],count,min_dist,next,i,j;
    for(i=0;i<num;i++)            //Assigning the values (initialisation)
    {
        for(j=0;j<num;j++)
        {
            if(G[i][j]==0)
            {
                cost[i][j]=INF;
            }
            else
            {
                cost[i][j]=G[i][j];
            }
        }
    }
    for(i=0;i<num;i++)
    {
        distance[i]=cost[start][i];
        pred[i]=start;
        visited[i] = 0;
    }
    distance[start] = 0;
    visited[start] = 1;
    count = 1;
    while(count < num-1)
    {
        min_dist=INF;
        for(i=0;i<num;i++)
        {
            if(distance[i]<min_dist && !visited[i])
            {
                min_dist=distance[i];
                next = i;
            }
        }
        visited[next] = 1;
        for(i=0;i<num;i++)
        {
            if(!visited[i])
            {
                if(min_dist+cost[next][i]<distance[i])        //Relax function
                {
                    distance[i]=min_dist+cost[next][i];
                    pred[i]=next;
                }
            }
        }
        count++;
    }
    cout<<endl;
    cout<<"Vertex"<<"           "<<"Distance"<<endl;
    cout<<"------------------------------------"<<endl;
    for(i=0;i<num;i++)
    {
        //if(i!=start)
        {
            cout<<i<<"                 "<<distance[i]<<endl;
            cout<<endl;
        }
    }
}
int main()
{
    int G[V][V];
    int source;
    cout<<"--------------------------DIJKSTRA'S ALGORTIHM-----------------------"<<endl;
    for(int i = 0; i <  V; i++)
    {
        cout<<"Enter the distance from vertex "<< i <<" to each vertex : "<<" ";
        for(int j = 0;j < V; j++)
        {
            cin>>G[i][j];
        }
    }
    cout<<endl<<endl;
    cout<<"Choose any vertex as source : "<<" ";
    cin>>source;
    dijkstra(G,V,source);
    return 0;
}
```

SAMPLE INPUT-OUTPUT



```
--------------------------DIJKSTRA'S ALGORTIHM-----------------------
Enter the distance from vertex 0 to each vertex :  0 10 3 0 0
Enter the distance from vertex 1 to each vertex :  0  0 1 2 0
Enter the distance from vertex 2 to each vertex :  0  4 0 8 2
Enter the distance from vertex 3 to each vertex :  0  0 0 0 7
Enter the distance from vertex 4 to each vertex :  0  0 0 9 0


Choose any vertex as source :  0

Vertex                 Distance
------------------------------------
0                  0

1                  7

2                  3

3                  9

4                  5
```

BELLMAN FORD

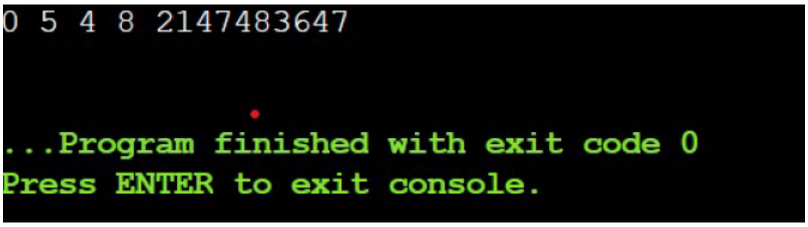
AIM

To implement Bellman Ford's Algorithm

PROGRAM

```
#include <bits/stdc++.h>
using namespace std;
// Struct for the edges of the graph
struct Edge {
    int u; //start vertex of the edge
    int v; //end vertex of the edge
    int w; //w of the edge (u,v)
};
struct Graph {
    int V; // Total number of vertices in the graph
    int E; // Total number of edges in the graph
    struct Edge* edge; // Array of edges
};
struct Graph* createGraph(int V, int E) {
    struct Graph* graph = new Graph;
    graph->V = V; // Total Vertices
    graph->E = E; // Total edges
    graph->edge = new Edge[E];
    return graph;
}
void printArr(int arr[], int size) {
    int i;
    for (i = 0; i < size; i++) {
    printf("%d ", arr[i]);
    }
    cout<<endl;
}
void BellmanFord(struct Graph* graph, int u) {
    int V = graph->V;
    int E = graph->E;
    int dist[V];
    // Step 1: fill the distance array and predecessor array
    for (int i = 0; i < V; i++)
    dist[i] = INT_MAX;
    // Mark the source vertex
    dist[u] = 0;
    // Step 2: relax edges |V| - 1 times
    for (int i = 1; i <= V - 1; i++) {
        for (int j = 0; j < E; j++) {
        // Get the edge data
        int u = graph->edge[j].u;
        int v = graph->edge[j].v;
        int w = graph->edge[j].w;
        if (dist[u] != INT_MAX && dist[u] + w < dist[v])
        dist[v] = dist[u] + w;
        }
    }
    // Step 3: detect negative cycle
    // if value changes then we have a negative cycle in the graph
    // and we cannot find the shortest distances
    for (int i = 0; i < E; i++) {
        int u = graph->edge[i].u;
        int v = graph->edge[i].v;
        int w = graph->edge[i].w;
        if (dist[u] != INT_MAX && dist[u] + w < dist[v]) {
        printf("Graph contains negative w cycle");
        return;
        }
    }
    // No negative weight cycle found!
    // Print the distance and predecessor array
    printArr(dist, V);
    return;
}
int main() {
    // Create a graph
    int V = 5; // Total vertices
    int E = 8; // Total edges
    struct Graph* graph = createGraph(V, E);
    //------- adding the edges of the graph
    //edge 0 --> 1
    graph->edge[0].u = 0;
    graph->edge[0].v = 1;
    graph->edge[0].w = 5;
    //edge 0 --> 2
    graph->edge[1].u = 0;
    graph->edge[1].v = 2;
    graph->edge[1].w = 4;
    //edge 1 --> 3
    graph->edge[2].u = 1;
    graph->edge[2].v = 3;
    graph->edge[2].w = 3;
    //edge 2 --> 1
    graph->edge[3].u = 2;
    graph->edge[3].v = 1;
    graph->edge[3].w = 6;
    //edge 3 --> 2
    graph->edge[4].u = 3;
    graph->edge[4].v = 2;
    graph->edge[4].w = 2;
    BellmanFord(graph, 0); //0 is the source vertex
    return 0;
}
```

SAMPLE INPUT-OUTPUT



```
0 5 4 8 2147483647


...Program finished with exit code 0
Press ENTER to exit console.
```

FLOYD WARSHALL

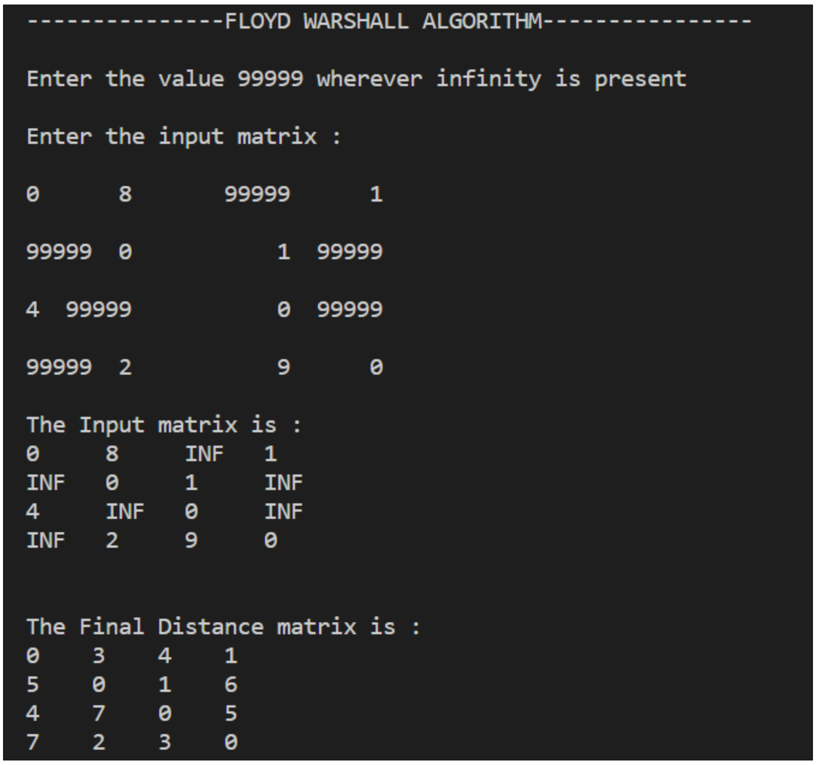
AIM

Program to implement All-Pairs Shortest Problem using Floyd's algorithm

PROGRAM

```
#include<iostream>
using namespace std;
#define INF 99999
#define num 4
void floyd_warshall(int A[][num])
{
    int i,j,k;
    for(k = 0;k<num;k++)
    {
        for(i= 0;i<num;i++)
        {
            for(j=0;j<num;j++)
            {
                if(A[i][j] > (A[i][k] + A[k][j]) && A[k][j]!= INF && A[i][k] != INF)
                {
                    A[i][j] = A[i][k] + A[k][j];

                }
            }
        }
    }
}

int main()
{
    int i,j;
    cout<<"---------------FLOYD WARSHALL ALGORITHM----------------"<<endl<<endl;
    cout<<"Enter the value 99999 wherever infinity is present "<<endl<<endl;
    cout<<"Enter the input matrix : "<<" ";
    int M[num][num];
    for(i=0;i<num;i++)
    {
        for(j=0;j<num;j++)
        {
            cin>>M[i][j];
        }
        cout<<endl;
    }
    cout<<"The Input matrix is : "<<endl;
    for(i=0;i<num;i++)
    {
        for(j=0;j<num;j++)
        {
            if (M[i][j] == INF)
            {
                cout<<"INF"<<"   ";
            }
            else
            {
                cout<< M[i][j]<<"     ";
            }
        }
        cout<<endl;
    }
    floyd_warshall(M);
    cout<<endl<<endl;
    cout<<"The Final Distance matrix is : "<<endl;
    for(i=0;i<num;i++)
    {
        for(j=0;j<num;j++)
        {
            if (M[i][j] == INF)
            {
                cout<<"INF"<<"   ";
            }
            else
            {
                cout<< M[i][j]<<"    ";
            }
        }
        cout<<endl;
    }
    return(0);
}
```

SAMPLE INPUT-OUTPUT



```
---------------FLOYD WARSHALL ALGORITHM----------------

Enter the value 99999 wherever infinity is present

Enter the input matrix :

0      8       99999      1

99999  0           1  99999

4  99999           0  99999

99999  2           9      0

The Input matrix is :
0     8     INF   1
INF   0     1     INF
4     INF   0     INF
INF   2     9     0


The Final Distance matrix is :
0    3    4    1
5    0    1    6
4    7    0    5
7    2    3    0
```

KRUSKAL'S ALGORITHM

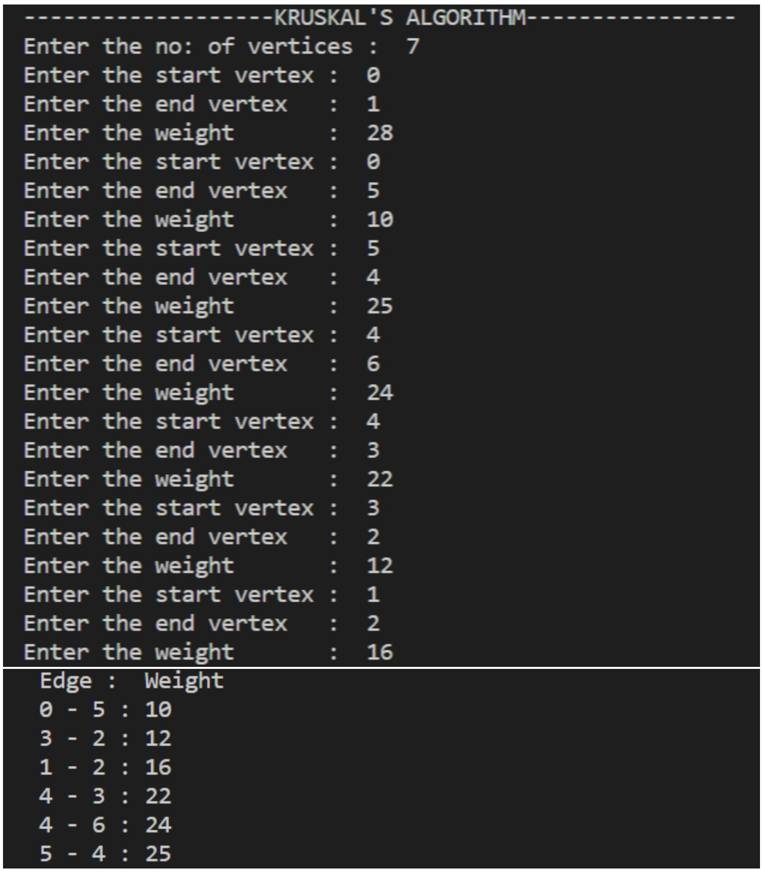
AIM

To find Minimal Cost Spanning Tree of a given undirected graph using Kruskal's algorithm

PROGRAM

```
// Kruskal's algorithm in C++
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
#define edge pair<int, int>
class Graph {
    private:
    vector<pair<int, edge> > G; // graph
    vector<pair<int, edge> > T; // mst
    int *parent;
    int V; // number of vertices/nodes in graph
    public:
    Graph(int V);
    void AddWeightedEdge(int u, int v, int w);
    int find_set(int i);
    void union_set(int u, int v);
    void kruskal();
    void print();
};
Graph::Graph(int V) {
    parent = new int[V];
//i 0 1 2 3 4 5
//parent[i] 0 1 2 3 4 5
    for (int i = 0; i < V; i++)
    parent[i] = i;
    G.clear();
    T.clear();
}
void Graph::AddWeightedEdge(int u, int v, int w) {
    G.push_back(make_pair(w, edge(u, v)));
}
int Graph::find_set(int i) {
    if (i == parent[i])
    return i;
    else
    return find_set(parent[i]);
}
void Graph::union_set(int u, int v) {
    parent[u] = parent[v];
}
void Graph::kruskal() {
    int i, uRep, vRep;
    sort(G.begin(), G.end()); // increasing weight
    for (i = 0; i < G.size(); i++) {
        uRep = find_set(G[i].second.first);
        vRep = find_set(G[i].second.second);
        if (uRep != vRep) {
            T.push_back(G[i]); // add to tree
            union_set(uRep, vRep);
        }
    }
}
void Graph::print() {
    cout << "Edge : "<< " Weight" << endl;
    for (int i = 0; i < T.size(); i++) {
    cout << T[i].second.first << " - " << T[i].second.second << " : "
    << T[i].first;
    cout << endl;
    }
}
int main() {
    int n,u,v,w;
    cout<<"-------------------KRUSKAL'S ALGORITHM----------------"<<endl;
    cout<<"Enter the no: of vertices : "<<" ";
    cin>>n;
    Graph g(n);
    for (int i = 0; i < n; i++)
    {
        cout<<"Enter the start vertex : "<<" ";
        cin>>u;
        cout<<"Enter the end vertex   : "<<" ";
        cin>>v;
        cout<<"Enter the weight       : "<<" ";
        cin>>w;
        g.AddWeightedEdge(u, v, w);
    }
    g.kruskal();
    g.print();
    return 0;
}
```

SAMPLE INPUT-OUTPUT



```
 -------------------KRUSKAL'S ALGORITHM----------------
 Enter the no: of vertices :  7
 Enter the start vertex :  0
 Enter the end vertex   :  1
 Enter the weight       :  28
 Enter the start vertex :  0
 Enter the end vertex   :  5
 Enter the weight       :  10
 Enter the start vertex :  5
 Enter the end vertex   :  4
 Enter the weight       :  25
 Enter the start vertex :  4
 Enter the end vertex   :  6
 Enter the weight       :  24
 Enter the start vertex :  4
 Enter the end vertex   :  3
 Enter the weight       :  22
 Enter the start vertex :  3
 Enter the end vertex   :  2
 Enter the weight       :  12
 Enter the start vertex :  1
 Enter the end vertex   :  2
 Enter the weight       :  16
  Edge :  Weight
  0 - 5 : 10
  3 - 2 : 12
  1 - 2 : 16
  4 - 3 : 22
  4 - 6 : 24
  5 - 4 : 25
```

PRIM'S ALGORITHM

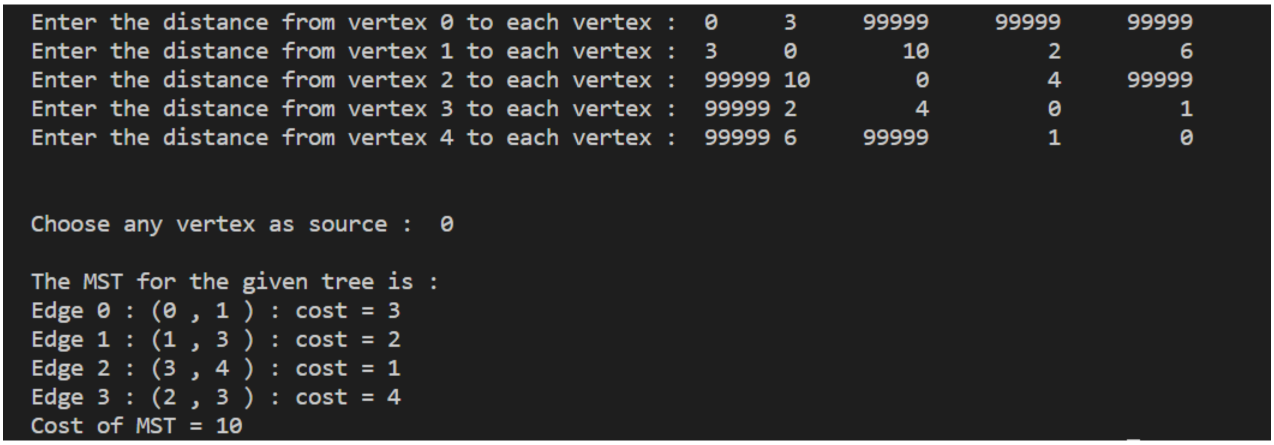
AIM

To find Minimal Cost Spanning Tree of a given undirected graph using Prim's algorithm

PROGRAM

```
#include <iostream>
#include <bits/stdc++.h>
using namespace std;
const int MAX = 99999; // INF
#define V 5
bool createsMST(int u, int v, vector<bool> V_MST)
{
    if (u == v)
    {
        return false;
    }
    if (V_MST[u] == false && V_MST[v] == false)
    {
        return false;
    }
    else if (V_MST[u] == true && V_MST[v] == true)
    {
        return false;
    }
    return true;
}
void MST_display(int cost[][V])
{
    vector<bool> V_MST(V, false);
    V_MST[0] = true;
    int edgeNo = 0, MSTcost = 0;
    while (edgeNo < V - 1)
    {
        int min = MAX, a = -1, b = -1;
        for (int i = 0; i < V; i++)
        {
            for (int j = 0; j < V; j++)
            {
                if (cost[i][j] < min)
                {
                    if (createsMST(i, j, V_MST))
                    {
                        min = cost[i][j];
                        a = i;
                        b = j;
                    }
                }
            }
        }
        if (a != -1 && b != -1)
        {
            cout << "Edge " << edgeNo++ << " : (" << a << " , " << b << " ) : cost = "
            << min << endl;
            MSTcost += min;
            V_MST[b] = V_MST[a] = true;
        }
    }
    cout << "Cost of MST = " << MSTcost;
}
int main()
{
    int G[V][V];
    int source;
    //Enter the value 99999 wherever infinity is present
    for (int i = 0; i < V; i++)
    {
        cout << "Enter the distance from vertex " << i << " to each vertex : "
             << " ";
        for (int j = 0; j < V; j++)
        {
            cin >> G[i][j];
        }
    }
    cout << endl<< endl;
    cout << "Choose any vertex as source : "<< " ";
    cin >> source;
    cout << endl;
    cout << "The MST for the given tree is :\n";
    cout<<endl;
    MST_display(G);
    return 0;
}
```

SAMPLE INPUT-OUTPUT



```
Enter the distance from vertex 0 to each vertex :  0     3     99999     99999     99999
Enter the distance from vertex 1 to each vertex :  3     0        10         2         6
Enter the distance from vertex 2 to each vertex :  99999 10        0         4     99999
Enter the distance from vertex 3 to each vertex :  99999 2         4         0         1
Enter the distance from vertex 4 to each vertex :  99999 6     99999         1         0


Choose any vertex as source :  0

The MST for the given tree is :
Edge 0 : (0 , 1 ) : cost = 3
Edge 1 : (1 , 3 ) : cost = 2
Edge 2 : (3 , 4 ) : cost = 1
Edge 3 : (2 , 3 ) : cost = 4
Cost of MST = 10
```

MATRIX CHAIN MULTIPLICATION

AIM

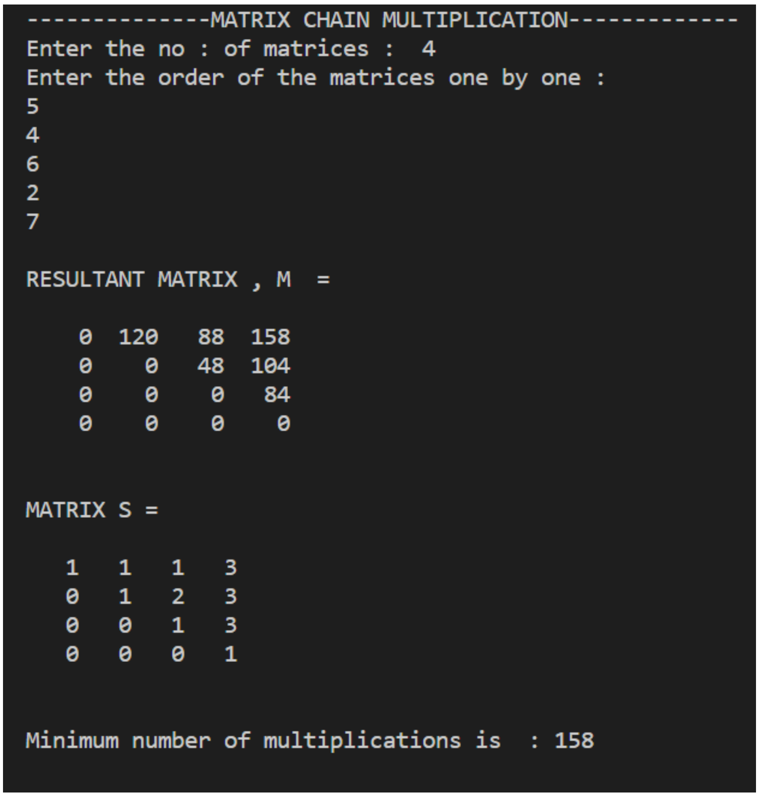
Program to implement Matrix Chain Multiplication using Dynamic Programming

PROGRAM

```
 #include <bits/stdc++.h>
#include <iostream>
#include <iomanip>
using namespace std;
void MatrixChainOrder(int p[], int n)
{
    int m[n][n];
    int s[n - 1][n - 1]; // Stores the value of k
    int i, j, k, L, q;

    for (i = 1; i < n; i++)
    {
        for (int j = 1; j < n; j++)
        {
            m[i][j] = 0;
        }
    }
    for (i = 1; i < n; i++)
    {
        for (int j = 1; j < n; j++)
        {
            if (i > j)
            {
                s[i][j] = 0;
            }
            else
            {
                s[i][j] = 1;
            }
        }
    }

    // L = chain length
    for (L = 2; L < n; L++)
    {
        for (i = 1; i < n - L + 1; i++)
        {
            j = i + L - 1;
            m[i][j] = INT_MAX;
            for (k = i; k <= j - 1; k++)
            {
                // q = cost/scalar multiplications
                q = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];
                if (q < m[i][j])
                {
                    m[i][j] = q;
                    s[i][j] = k;
                }
            }
        }
    }
    cout << "RESULTANT MATRIX , M  =  " << endl
         << endl;
    for (int i = 1; i < n; i++)
    {
        for (int j = 1; j < n; j++)
        {
            cout << setw(5) << m[i][j] << setw(4);
        }
        cout << endl;
    }
    cout << endl
         << endl;
    cout << "MATRIX S =  " << endl
         << endl;
    for (int i = 1; i < n; i++)
    {
        for (int j = 1; j < n; j++)
        {
            cout << setw(4) << s[i][j] << setw(4);
        }
        cout << endl;
    }
    cout<<endl<<endl;
    cout << "Minimum number of multiplications is  : " << m[1][n - 1]<<endl;
}
```

```
int main()
{
    cout<<"--------------MATRIX CHAIN MULTIPLICATION-------------"<<endl;
    int num;
    cout << "Enter the no : of matrices : "
         << " ";
    cin >> num;
    int A[num + 1];
    cout << "Enter the order of the matrices one by one : "
             << " ";
    for (int i = 0; i < num + 1; i++)
    {
        cin >> A[i];
    }
    int size = sizeof(A) / sizeof(A[0]);
    cout<<endl;
    MatrixChainOrder(A, size);
    cout<<endl<<endl;
    return 0;
}
```

SAMPLE INPUT-OUTPUT



```
--------------MATRIX CHAIN MULTIPLICATION-------------
Enter the no : of matrices :  4
Enter the order of the matrices one by one :
5
4
6
2
7

RESULTANT MATRIX , M  =

    0  120   88  158
    0    0   48  104
    0    0    0   84
    0    0    0    0


MATRIX S =

   1   1   1   3
   0   1   2   3
   0   0   1   3
   0   0   0   1


Minimum number of multiplications is  : 158
```

0/1 KNAPSACK PROBLEM

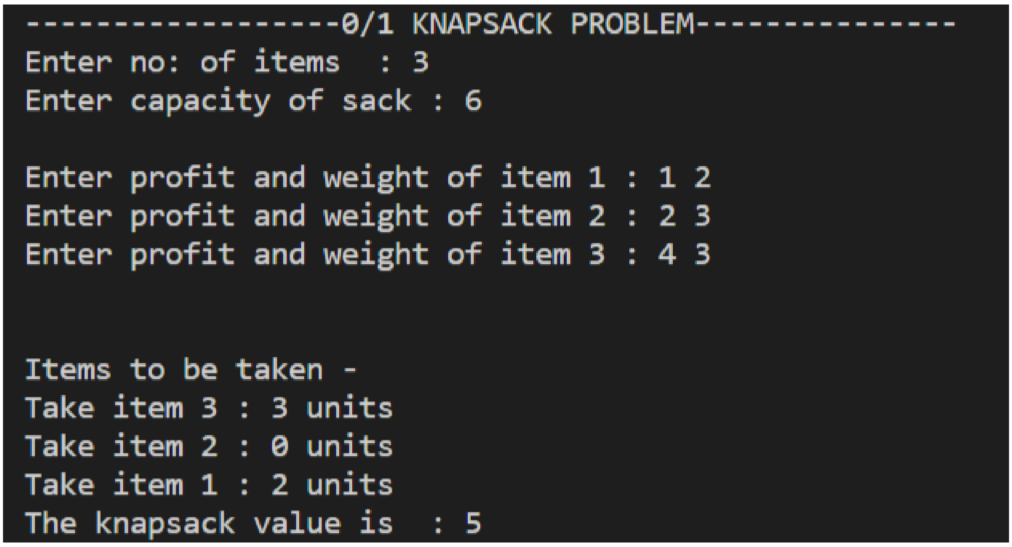
AIM

Program to implement 0/1 Knapsack Problem using Dynamic Programming

PROGRAM

```
#include<iostream>
#define MAX 10
using namespace std;
struct product
{
  int product_num;
  int profit;
  int weight;
  float ratio;
  float take_quantity;
};
int main()
{
  cout<<"------------------0/1 KNAPSACK PROBLEM---------------"<<endl;
  product P[MAX],temp;
  int i,j,total_product,capacity;
  float value=0;
  cout<<"Enter no: of items  : ";
  cin>>total_product;
  cout<<"Enter capacity of sack : ";
  cin>>capacity;
  cout<<"\n";
  for(i=0;i<total_product;++i)
  {
    P[i].product_num=i+1;
    cout<<"Enter profit and weight of item "<<i+1<<" : ";
    cin>>P[i].profit>>P[i].weight;

    P[i].ratio=(float)P[i].profit/P[i].weight;
    P[i].take_quantity=0;
  }
  //HIGHEST RATIO BASED SORTING
  for(i=0;i<total_product;++i)
  {
    for(j=i+1;j<total_product;++j)
    {
      if(P[i].ratio<P[j].ratio)
      {
        temp=P[i];
        P[i]=P[j];
        P[j]=temp;
      }
    }
  }
  for(i=0;i<total_product;++i)
  {
    if(capacity==0)
      break;
    else if(P[i].weight<capacity)
    {
      P[i].take_quantity=1;
      capacity-=P[i].weight;
    }
    else if(P[i].weight>capacity)
    {
      P[i].take_quantity=(float)capacity/P[i].weight;
      capacity=0;
    }
  }

  cout<<"\n\nItems to be taken -";
  for(i=0;i<total_product;++i)
  {
    cout<<"\nTake item "<<P[i].product_num<<" : "<<P[i].take_quantity*P[i].weight<<
    " units";
    value+=P[i].profit*P[i].take_quantity;
  }
  cout<<"\nThe knapsack value is  : "<<value;

}
```

SAMPLE INPUT-OUTPUT



```
------------------0/1 KNAPSACK PROBLEM---------------
Enter no: of items  : 3
Enter capacity of sack : 6

Enter profit and weight of item 1 : 1 2
Enter profit and weight of item 2 : 2 3
Enter profit and weight of item 3 : 4 3


Items to be taken -
Take item 3 : 3 units
Take item 2 : 0 units
Take item 1 : 2 units
The knapsack value is  : 5
```

HUFFMAN CODES

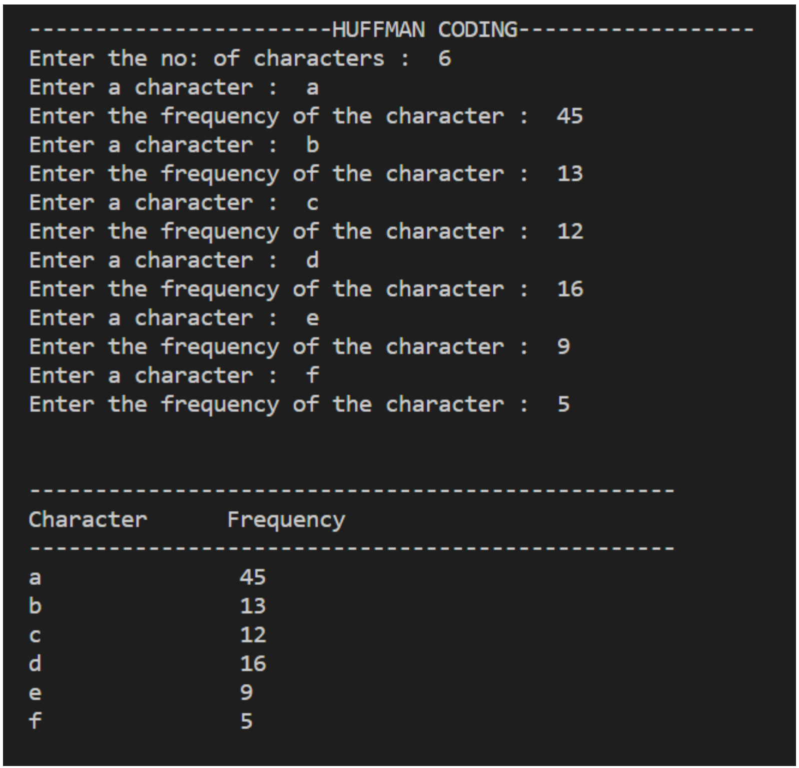
AIM

Program to implement Huffman Coding using Greedy algorithm

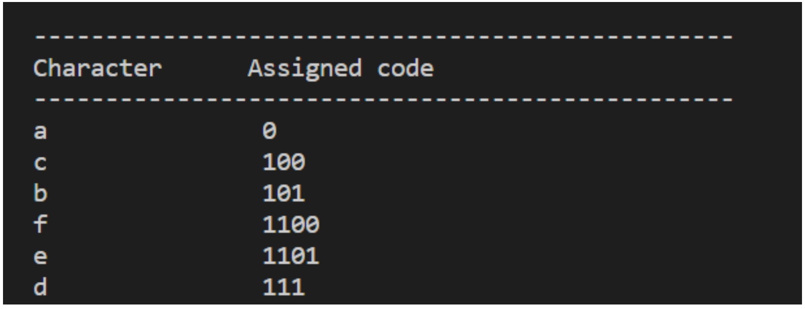
PROGRAM

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

struct MinHeapNode
{
    char d;
    unsigned frequency;
    MinHeapNode *lChild, *rChild;

    MinHeapNode(char d, unsigned frequency)

    {

        lChild = rChild = NULL;
        this->d = d;
        this->frequency = frequency;
    }
};

// function to compare
struct compare
{
    bool operator()(MinHeapNode *l, MinHeapNode *r)
    {
        return (l->frequency > r->frequency);
    }
};

void printCodes(struct MinHeapNode *root, string str)
{
    if (!root)
        return;
    if (root->d != ’$’)
        cout << root->d << "              " << str << "\n";
    printCodes(root->lChild, str + "0");
    printCodes(root->rChild, str + "1");
}

void HuffmanCodes(char d[], int frequency[], int size)
{
    struct MinHeapNode *lChild, *rChild, *top;

    priority_queue<MinHeapNode *, vector<MinHeapNode *>, compare> minHeap;

    for (int i = 0; i < size; i++)
        minHeap.push(new MinHeapNode(d[i], frequency[i]));

    while (minHeap.size() != 1)
    {
        lChild = minHeap.top();
        minHeap.pop();

        rChild = minHeap.top();
        minHeap.pop();

        top = new MinHeapNode('$', lChild->frequency + rChild->frequency);

        top->lChild = lChild;
        top->rChild = rChild;

        minHeap.push(top);
    }
    printCodes(minHeap.top(), " ");
}

int main()
{
    int num;
    cout<<"-----------------------HUFFMAN CODING------------------"<<endl;
    cout << "Enter the no: of characters : "<< " ";
    cin >> num;
    char A[num];
    int X[num];
    for (int i = 0; i < num; i++)
    {
        cout << "Enter a character : "<< " ";
        cin >> A[i];
        cout << "Enter the frequency of the character : "<< " ";
        cin >> X[i];
    }
    cout<<endl<<endl;
    cout<<"-------------------------------------------------"<<endl;
    cout<<"Character"<<"     "<<"Frequency"<<endl;
    cout<<"-------------------------------------------------"<<endl;
    for (int i = 0; i < num; i++)
    {
        cout<< A[i]<<"               "<<X[i]<<endl;
    }
    int size = sizeof(A) / sizeof(A[0]);
    cout<<endl;
    cout<<"-------------------------------------------------"<<endl;
    cout<<"Character"<<"     "<<"Assigned code"<<endl;
    cout<<"-------------------------------------------------"<<endl;
    HuffmanCodes(A, X, size);
    return 0;
}
```

SAMPLE INPUT-OUTPUT



```
-----------------------HUFFMAN CODING------------------
Enter the no: of characters :  6
Enter a character :  a
Enter the frequency of the character :  45
Enter a character :  b
Enter the frequency of the character :  13
Enter a character :  c
Enter the frequency of the character :  12
Enter a character :  d
Enter the frequency of the character :  16
Enter a character :  e
Enter the frequency of the character :  9
Enter a character :  f
Enter the frequency of the character :  5


-------------------------------------------------
Character      Frequency
-------------------------------------------------
a               45
b               13
c               12
d               16
e               9
f               5
```



```
-------------------------------------------------
Character      Assigned code
-------------------------------------------------
a               0
c               100
b               101
f               1100
e               1101
d               111
```

TRAVELLING SALESMAN PROBLEM

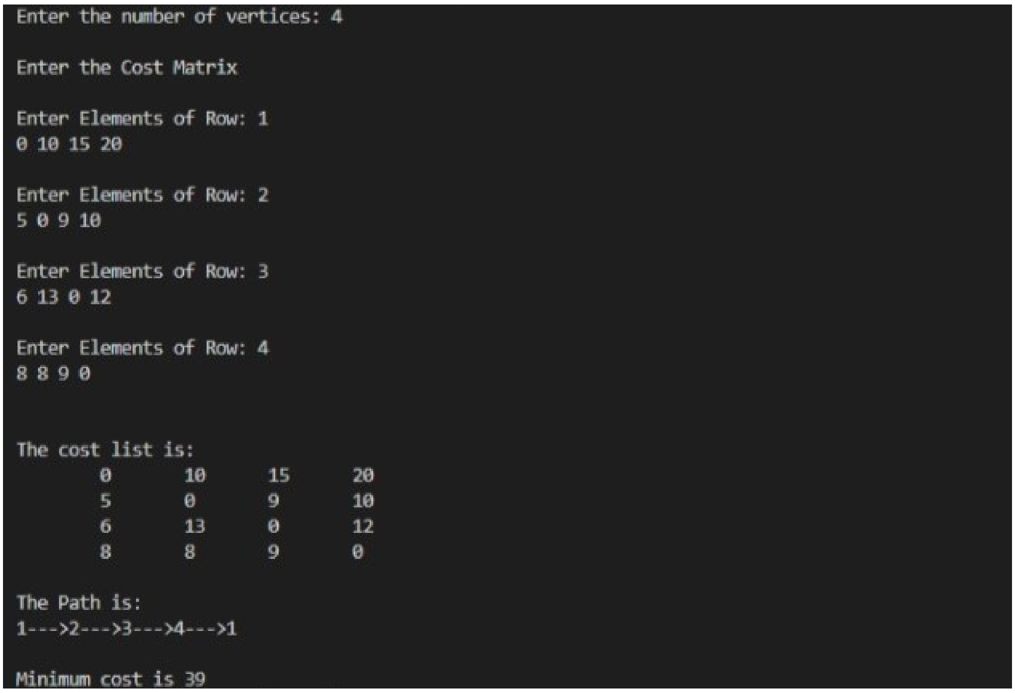
AIM

Program to find the optimal solution for the Travelling Salesman Problem

PROGRAM

```
#include<iostream>
using namespace std;
int ary[10][10],completed[10],n,cost=0;
void takeInput()
{
    int i,j;
    cout<<"Enter the number of vertices: ";
    cin>>n;
    cout<<"\nEnter the Cost Matrix\n";
    for(i=0;i < n;i++)
    {
        cout<<"\nEnter Elements of Row: "<<i+1<<"\n";
        for( j=0;j < n;j++)
            cin>>ary[i][j];
            completed[i]=0;
    }
    cout<<"\n\nThe cost list is:";
    for( i=0;i < n;i++)
    {
        cout<<"\n";
        for(j=0;j < n;j++)
        cout<<"\t"<<ary[i][j];
    }
}
int least(int c)
{
    int i,nc=999;
    int min=999,kmin;
    for(i=0;i < n;i++)
    {
        if((ary[c][i]!=0)&&(completed[i]==0))
        if(ary[c][i]+ary[i][c] < min)
        {
            min=ary[i][0]+ary[c][i];
            kmin=ary[c][i];
            nc=i;
        }
    }
    if(min!=999)
        cost+=kmin;
    return nc;
}
void mincost(int city)
{
    int i,ncity;
    completed[city]=1;
    cout<<city+1<<"--->";
    ncity=least(city);
    if(ncity==999)
    {
        ncity=0;
        cout<<ncity+1;
        cost+=ary[city][ncity];
        return;
    }
    mincost(ncity);
}
int main()
{
    takeInput();
    cout<<"\n\nThe Path is:\n";
    mincost(0); //passing 0 because starting vertex
    cout<<"\n\nMinimum cost is "<<cost;
    return 0;
}
```

SAMPLE INPUT-OUTPUT



```
Enter the number of vertices: 4

Enter the Cost Matrix

Enter Elements of Row: 1
0 10 15 20

Enter Elements of Row: 2
5 0 9 10

Enter Elements of Row: 3
6 13 0 12

Enter Elements of Row: 4
8 8 9 0


The cost list is:
        0       10      15      20
        5       0       9       10
        6       13      0       12
        8       8       9       0

The Path is:
1--->2--->3--->4--->1

Minimum cost is 39
```