

**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

**ABDUL HAKKEEM P A
(80521001)**

**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 **ABDUL HAKKEEM P A (80521001)**, 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

Raheena Salihin
Guest Faculty
Department of Computer Science
CUSAT

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

Table of Contents

Sl.No.	Program	Pg.No.
1	Program to sort elements using Quick Sort	pg 1
2	Program to print all nodes reachable from a graph using BFS.	pg 3
3	Program to check whether a given graph is connected or not using DFS method	pg 5
4	Program to find graph, find shortest paths to other vertices using Dijkstra's algorithm	pg 7
5	Program to implement Bellman Ford's Algorithm.	pg 9
6	Program to find the transitive closure of a given directed graph using Floyd Warshall's algorithm	pg 12
7	Program to find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm	pg 14
8	Program to find Minimum Cost Spanning Tree of a given undirected graph using Prim's algorithm.	pg 17
9	Program to Implement matrix chain multiplication using dynamic programming	pg 20
10	Program to implement 0/1 Knapsack problem using Dynamic Programming.	pg 23
11	Program to implement Huffman coding using Greedy algorithm	pg 25
12	Program to find optimal solution to Travelling Salesman Problem	pg 28

QUICK SORT

AIM

Sort a given set of elements using the Quicksort method

PROGRAM

```
#include <iostream>
#include <bits/stdc++.h>
#include <cstdlib>

using namespace std;

int partition(int *data,int begin,int end);

void quickSort(int *data,int begin,int end){
    int loc;
    if(begin<end){
        loc = partition(data,begin,end);
        quickSort(data,begin,loc-1);
        quickSort(data,loc+1,end);
    }
}

int partition(int *data,int begin,int end){
    int pivot = data[begin];
    int lowerBound = begin;
    int upperBound = end;
    int temp;
    while(lowerBound<upperBound){
        while(data[lowerBound]<=pivot){
            lowerBound++;
        }
        while(data[upperBound]>pivot){
            upperBound--;
        }
        if(lowerBound<upperBound){
            temp = data[lowerBound];
            data[lowerBound]=data[upperBound];
            data[upperBound]=temp;
        }
    }
}
```

```
    }
    temp = data[begin];
    data[begin] = data[upperBound];
    data[upperBound] = temp;
    return upperBound;
}

void printArray(int* data,int length){
for (int var = 0; var < length; ++var) {
cout<<data[var]<<" ";
}
}

int main(){

    int array[20];
    srand(time(0));
    for (int i = 0; i <20;i++){
        array[i] = rand();
    }
    clock_t start, end;
    start = clock();
    int length = sizeof(array)/sizeof(array[0]);
    quickSort(array,0,length-1);
    end = clock();
    double time_taken = double(end - start) / double(CLOCKS_PER_SEC);
    cout << "Time taken by program is : " << fixed
    << time_taken << setprecision(5);
    cout << " sec " << endl;
    cout<<endl;
    printArray(array,length);
}
```

SAMPLE INPUT-OUTPUT

```
Enter the no of elements : 5
Value 1 : 50
Value 2 : 10
Value 3 : 30
Value 4 : 40
Value 5 : 20
Input Data : 50,10,30,40,20,
```

```
10 20 30 40 50
```

BREADTH FIRST SEARCH

AIM

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

PROGRAM

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

class Graph {
    int V;
    vector<list<int> > adj;

public:
    Graph(int V);
    void addEdge(int v, int w);
    void BFS(int s);
};

Graph::Graph(int V)
{
    this->V = V;
    adj.resize(V);
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

void Graph::BFS(int s)
{
    vector<bool> visited;
    visited.resize(V, false);

    list<int> queue;

    visited[s] = true;
    queue.push_back(s);
```

```
while (!queue.empty()) {
    s = queue.front();
    cout << s << " ";
    queue.pop_front();

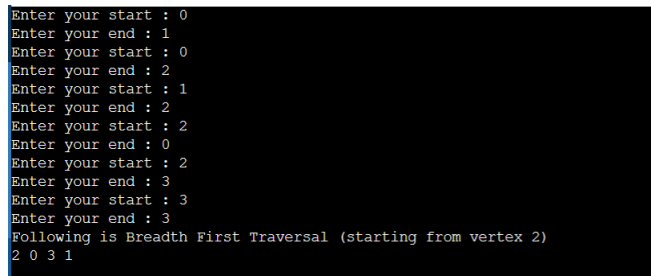
    for (auto adjacent : adj[s]) {
        if (!visited[adjacent]) {
            visited[adjacent] = true;
            queue.push_back(adjacent);
        }
    }
}

int main()
{
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Breadth First Traversal "
         << "(starting from vertex 2) \n";
    g.BFS(2);

    return 0;
}
```

SAMPLE INPUT-OUTPUT



```
Enter your start : 0
Enter your end : 1
Enter your start : 0
Enter your end : 2
Enter your start : 1
Enter your end : 2
Enter your start : 2
Enter your end : 0
Enter your start : 2
Enter your end : 3
Enter your start : 3
Enter your end : 3
Following is Breadth First Traversal (starting from vertex 2)
2 0 3 1
```

DEPTH FIRST SEARCH

AIM

Check whether a given graph is connected or not using DFS method

PROGRAM

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

class Graph {
public:
    map<int, bool> visited;
    map<int, list<int> > adj;

    void addEdge(int v, int w);
    void DFS(int v);
};

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

void Graph::DFS(int v)
{
    visited[v] = true;
    cout << v << " ";

    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFS(*i);
}

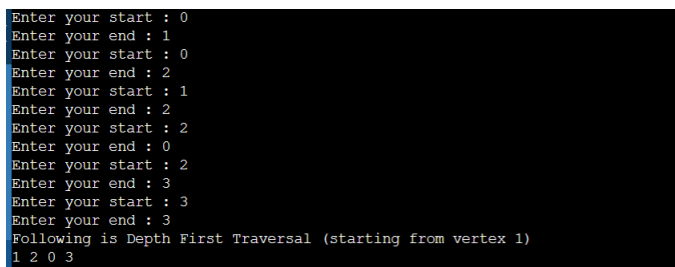
int main()
{
    Graph g;
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
}
```



```
g.addEdge(2, 0);
g.addEdge(2, 3);
g.addEdge(3, 3);

cout << "Following is Depth First Traversal"
      << " (starting from vertex 1) \n";
g.DFS(1);
return 0;
}
```

SAMPLE INPUT-OUTPUT



```
Enter your start : 0
Enter your end : 1
Enter your start : 0
Enter your end : 2
Enter your start : 1
Enter your end : 2
Enter your start : 2
Enter your end : 0
Enter your start : 2
Enter your end : 3
Enter your start : 3
Enter your end : 3
Following is Depth First Traversal (starting from vertex 1)
1 2 0 3
```

DIJKSTRA'S ALGORITHM

AIM

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

PROGRAM

```
#include <iostream>
using namespace std;
#include <limits.h>

#define V 9

int minDistance(int dist[], bool sptSet[])
{
    int min = INT_MAX, min_index;
    for (int v = 0; v < V; v++)
        if (sptSet[v] == false && dist[v] <= min)
            min = dist[v], min_index = v;

    return min_index;
}

void printSolution(int dist[])
{
    cout << "Vertex \t Distance from Source" << endl;
    for (int i = 0; i < V; i++)
        cout << i << " \t\t\t" << dist[i] << endl;
}

void dijkstra(int graph[V][V], int src)
{
    int dist[V];

    bool sptSet[V];
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = false;

    dist[src] = 0;
```

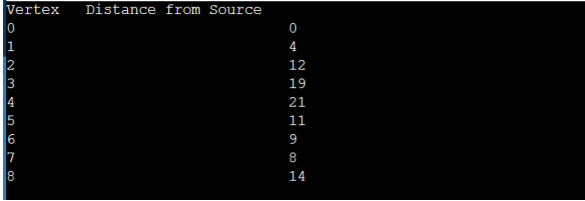
```
for (int count = 0; count < V - 1; count++) {
    int u = minDistance(dist, sptSet);
    sptSet[u] = true;
    for (int v = 0; v < V; v++)

        if (!sptSet[v] && graph[u][v]
            && dist[u] != INT_MAX
            && dist[u] + graph[u][v] < dist[v])
            dist[v] = dist[u] + graph[u][v];
}
printSolution(dist);
}

int main()
{
    int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
                        { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
                        { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
                        { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
                        { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
                        { 0, 0, 4, 14, 10, 0, 2, 0, 0 },
                        { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
                        { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
                        { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };

    dijkstra(graph, 0);
    return 0;
}
```

SAMPLE INPUT-OUTPUT



Vertex	Distance from Source
0	0
1	4
2	12
3	19
4	21
5	11
6	9
7	8
8	14

BELLMAN FORD ALGORITHM

AIM

Implement Bellman Ford's Algorithm.

PROGRAM

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

struct Edge {
    int src, dest, weight;
};

struct Graph {
    int V, E;
    struct Edge* edge;
};

struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = new Graph;
    graph->V = V;
    graph->E = E;
    graph->edge = new Edge[E];
    return graph;
}

void printArr(int dist[], int n)
{
    printf("Vertex\t\tDistance from Source\n");
    for (int i = 0; i < n; ++i)
        printf("%d \t\t %d\n", i, dist[i]);
}

void BellmanFord(struct Graph* graph, int src)
{
    int V = graph->V;
    int E = graph->E;
    int dist[V];
```

```
for (int i = 0; i < V; i++)
dist[i] = INT_MAX;
dist[src] = 0;

for (int i = 1; i <= V - 1; i++) {
for (int j = 0; j < E; j++) {
int u = graph->edge[j].src;
int v = graph->edge[j].dest;
int weight = graph->edge[j].weight;
if (dist[u] != INT_MAX
&& dist[u] + weight < dist[v])
dist[v] = dist[u] + weight;
}
}

for (int i = 0; i < E; i++) {
int u = graph->edge[i].src;
int v = graph->edge[i].dest;
int weight = graph->edge[i].weight;
if (dist[u] != INT_MAX
&& dist[u] + weight < dist[v]) {
printf("Graph contains negative weight cycle");
return; // If negative cycle is detected, simply
// return
}
}

printArr(dist, V);
return;
}

// Driver's code
int main()
{
int V = 5; // Number of vertices in graph
int E = 8; // Number of edges in graph
struct Graph* graph = createGraph(V, E);

graph->edge[0].src = 0;
graph->edge[0].dest = 1;
```

```
graph->edge[0].weight = -1;

graph->edge[1].src = 0;
graph->edge[1].dest = 2;
graph->edge[1].weight = 4;

graph->edge[2].src = 1;
graph->edge[2].dest = 2;
graph->edge[2].weight = 3;

graph->edge[3].src = 1;
graph->edge[3].dest = 3;
graph->edge[3].weight = 2;

graph->edge[4].src = 1;
graph->edge[4].dest = 4;
graph->edge[4].weight = 2;

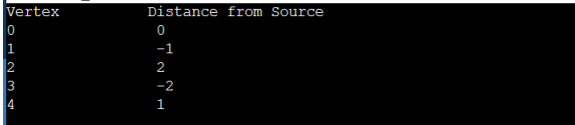
graph->edge[5].src = 3;
graph->edge[5].dest = 2;
graph->edge[5].weight = 5;

graph->edge[6].src = 3;
graph->edge[6].dest = 1;
graph->edge[6].weight = 1;

graph->edge[7].src = 4;
graph->edge[7].dest = 3;
graph->edge[7].weight = -3;

BellmanFord(graph, 0);
return 0;
}
```

SAMPLE INPUT-OUTPUT



Vertex	Distance from Source
0	0
1	-1
2	2
3	-2
4	1

FLOYD WARSHALL'S ALGORITHM

AIM

Compute the transitive closure of a given directed graph using Floyd Warshall's algorithm.

PROGRAM

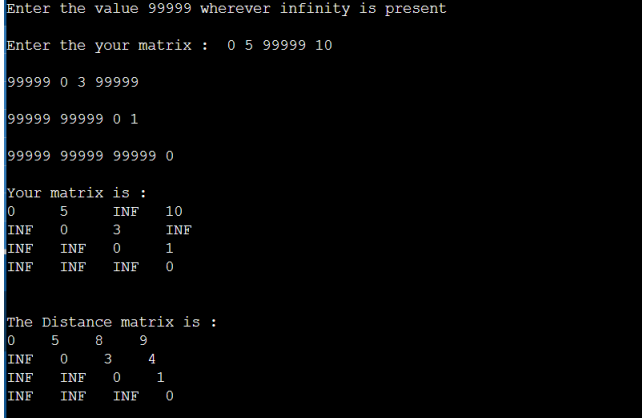
```
#include<iostream>
using namespace std;
#define INF 99999
#define num 4

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,no_of_vertices;
    cout<<"Enter the value 99999 wherever infinity is present "<<endl<<endl;
    cout<<"Enter the your matrix : "<<" ";
    int M[num][num];
    for(i=0;i<num;i++){
        for(j=0;j<num;j++){
            cin>>M[i][j];
        }
        cout<<endl;
    }
    cout<<"Your 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 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



```
Enter the value 99999 wherever infinity is present
Enter the your matrix :  0 5 99999 10
99999 0 3 99999
99999 99999 0 1
99999 99999 99999 0

Your matrix is :
0      5      INF    10
INF    0      3      INF
INF    INF    0      1
INF    INF    INF    0

The Distance matrix is :
0      5      8      9
INF    0      3      4
INF    INF    0      1
INF    INF    INF    0
```


KRUSKAL'S ALGORITHM

AIM

Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm..

PROGRAM

```
#include <iostream>
#include <bits/stdc++.h>

using namespace std;
const int MAX = 99999;
int parent[MAX];
int find(int a)
{
    while (parent[a] != a)
    {
        parent[a] = parent[parent[a]];
        a = parent[a];
    }
    return a;
}

void add(int a, int b)
{
    int d = find(a);
    int e = find(b);
    parent[d] = parent[e];
}

int main()
{
    int V, E;
    cout << "Enter the no: of vertices : "<< " ";
    cin >> V;
    cout << "Enter the no: of edges : "<< " ";
    cin >> E;
    vector<pair<int, pair<int, int>>> adj;
    cout << "Enter the weight,source and destination one by one in correct order
: " << endl;
    for (int i = 0; i < E; i++)
```

```
{
    int weight;
    int src, destination;
    cin >> weight >> src >> destination;
    adj.push_back({weight, {src, destination}});
}
sort(adj.begin(), adj.end());
for (int i = 0; i < MAX; i++)
{
    parent[i] = i;
}
vector<pair<int, int>> tree_edges; // Storing the edges of MST
int totalweight = 0;
for (auto x : adj)
{
    int a = x.second.first;
    int b = x.second.second;
    int cost = x.first;
    if (find(a) != find(b))
    {
        totalweight += cost;
        add(a, b);
        tree_edges.push_back({a, b});
    }
}
cout << "Edges are : " << endl;
for (auto x : tree_edges) // MST edges
{
    cout << x.first << " " << x.second << endl;
}
cout << "Total weight of MST = ";
cout << totalweight << endl;

return (0);
}
```

SAMPLE INPUT-OUTPUT

```
Enter the no: of vertices : 4
Enter the no: of edges : 5
Enter the weight,source and destination one by one in correct order :
10 0 1
15 1 3
4 2 3
6 2 0
5 0 3
Edges are :
2 3
0 3
0 1
Total weight of MST = 19
```

PRIM'S ALGORITHM

AIM

Find Minimum 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;
    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];
        }
    }

    for (int i = 0; i < V; i++) // Assigning the values (initialisation)
    {
        for (int j = 0; j < V; j++)
        {
            if (G[i][j] == -1)
            {
```

```
        G[i][j] = MAX;
    }
    else
    {
        G[i][j] = 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";
MST_display(G);
return 0;
}
```

SAMPLE INPUT-OUTPUT

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

Choose any vertex as source : 1

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

Implement matrix chain multiplication using dynamic programming

PROGRAM

```
#include <bits/stdc++.h>
#include <iostream>
#include <iomanip>
using namespace std;
int 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++)
```

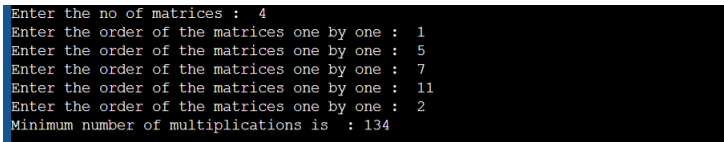
```
{
    for (i = 1; i < n - L + 1; i++)
    {
        j = i + L - 1;
        m[i][j] = INT_MAX;
        for (k = i; k <= j - 1; k++)
        {
            // q = cost/scalar multiplications
            q = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];
            if (q < m[i][j])
            {
                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;
}
return m[1][n - 1];
}
```



```
int main()
{
    // int A[] = {5, 4, 6, 2, 7};
    int num;
    cout << "Enter the no : of matrices : "
         << " ";
    cin >> num;
    int A[num + 1];
    for (int i = 0; i < num + 1; i++)
    {
        cout << "Enter the order of the matrices one by one : "
             << " ";
        cin >> A[i];
    }
    int size = sizeof(A) / sizeof(A[0]);
    cout << "Minimum number of multiplications is : " <<
    MatrixChainOrder(A, size);
    return 0;
}
```

SAMPLE INPUT-OUTPUT

A screenshot of a terminal window showing the program's output. The text is as follows:

```
Enter the no of matrices : 4
Enter the order of the matrices one by one : 1
Enter the order of the matrices one by one : 5
Enter the order of the matrices one by one : 7
Enter the order of the matrices one by one : 11
Enter the order of the matrices one by one : 2
Minimum number of multiplications is : 134
```

0/1 KNAPSACK PROBLEM

AIM

Implement 0/1 Knapsack problem using Dynamic Programming.

PROGRAM

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

int max(int a, int b) { return (a > b) ? a : b; }

int knapSack(int W, int wt[], int val[], int n)
{
    if (n == 0 || W == 0)
        return 0;

    if (wt[n - 1] > W)
        return knapSack(W, wt, val, n - 1);

    else
        return max(
            val[n - 1]
            + knapSack(W - wt[n - 1], wt, val, n - 1),
            knapSack(W, wt, val, n - 1));
}

// Driver code
int main()
{
    int val[] = { 60, 100, 120 };
    int wt[] = { 10, 20, 30 };
    int W = 50;
    int n = sizeof(val) / sizeof(val[0]);
    cout<<"Total Weight : "<<W<<endl;
    cout<<"Weights : ";
    for (int i = 0; i < 3; i++) {
        cout<<wt[i]<<" ";
    }
    cout<<"\nValues : ";
    for (int i = 0; i < 3; i++) {
```

```
        cout<<val[i]<<" ";
    }
    cout <<"\nThe maximum value of items that can be put into knapsack is
        : "<< knapSack(W, wt, val, n);
    return 0;
}
```

SAMPLE INPUT-OUTPUT

```
Total Weight : 50
Weights : 10 20 30
Values : 60 100 120
The maximum value of items that can be put into knapsack is :220
```

HUFFMAN CODING

AIM

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 << "Enter the no: of characters : "
         << " ";
    cin >> num;
    char A[num];
```

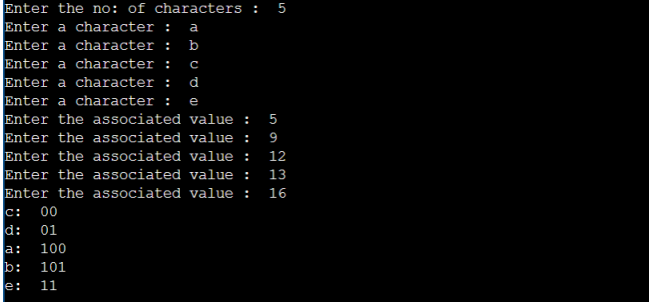
```
int X[num];
for (int i = 0; i < num; i++)
{
    cout << "Enter a character : "
          << " ";
    cin >> A[i];
}
for (int i = 0; i < num; i++)
{
    cout << "Enter the associated value : "
          << " ";
    cin >> X[i];
}

int size = sizeof(A) / sizeof(A[0]);

HuffmanCodes(A, X, size);

return 0;
}
```

SAMPLE INPUT-OUTPUT



```
Enter the no: of characters : 5
Enter a character : a
Enter a character : b
Enter a character : c
Enter a character : d
Enter a character : e
Enter the associated value : 5
Enter the associated value : 9
Enter the associated value : 12
Enter the associated value : 13
Enter the associated value : 16
c: 00
d: 01
a: 100
b: 101
e: 11
```

TRAVELLING SALESMAN PROBLEM

AIM

Implement any scheme to find the optimal solution for the Traveling Salesman Person problem and then solve the same problem instance using any approximation algorithm and determine the error in the approximation.

PROGRAM

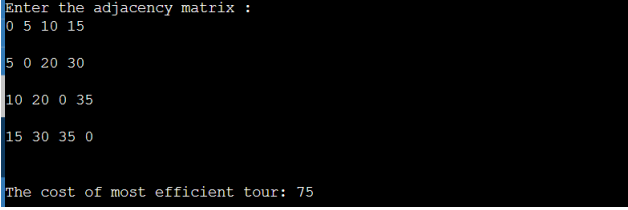
```
#include <iostream>
#include <bits/stdc++.h>

using namespace std;
#define vr 4
int TSP(int graph[][vr], int p)
{
    vector<int> ver;
    for (int i = 0; i < vr; i++)
    {
        if (i != p)
            ver.push_back(i);
    }
    int m_p = INT_MAX;
    do
    {
        int current_path = 0;
        int k = p;
        for (int i = 0; i < ver.size(); i++)
        {
            current_path += graph[k][ver[i]];
            k = ver[i];
        }
        current_path += graph[k][p];
        m_p = min(m_p, current_path); // to update the value of minimum weight
    } while (next_permutation(ver.begin(), ver.end()));
    return m_p;
}

int main()
{
    int graph[vr][vr];
    cout << "Enter the adjacency matrix : " << endl;
```

```
    for (int i = 0; i < vr; i++){
        for (int j = 0; j < vr; j++){
            cin >> graph[i][j];
        }
        cout << endl;
    }
    int p = 0;
    cout<<"\nThe cost of most efficient tour: " << TSP(graph, p) << endl;
    return 0;
}
```

SAMPLE INPUT-OUTPUT



```
Enter the adjacency matrix :
0 5 10 15
5 0 20 30
10 20 0 35
15 30 35 0
The cost of most efficient tour: 75
```