***DISCRETE MATHEMATICS ASSIGNMENT***

*Submitted By:*

*NAME:- YASHVI PATEL*

*ROLL NO. :- 17CS6004*

*NAME:-KUNDAN KUMAR*

*ROLL NO. :- 17CS6005*

*NAME:-SAMEEKSHA KESHARI*

*ROLL NO. :-17CS6006*

COURCE NAME :- DISCRETE MATHEMATICS

COURCE CODE :- CS1101

SUBMISSION DATE :- 11/11/2017

**Contents**

|  |  |  |
| --- | --- | --- |
| **1** | **Breadth First Search (BFS)** | **1-4** |
| **2** | **Depth First Search (DFS)** | **4-7** |
| **3** | **Dijkstra’s Shortest Path** | **8-11** |
| **4** | **Centre of the Graph** | **11-15** |
| **5** | **Euler’s Path** | **15-23** |
| **6** | **Hamiltonian Cycle** | **24-29** |
| **7** | **Travelling Sales Man Problem** | **29-33** |
| **8** | **Relation Matrix** | **33-38** |

DISCRETE MATHEMATICS

***ASSIGNMENT***

**BFS**

**Logic :**

There are many ways to traverse graphs. BFS is the most commonly used approach.

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.

As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

1. First move horizontally and visit all the nodes of the current layer
2. Move to the next layer

**Algorithm:**

BFS (G, s) : // Where G is the graph and s is the source node

let Q be queue.

Q.enqueue( s ) //Inserting s in queue until all its neighbour vertices are marked.

mark s as visited.

while ( Q is not empty)

// Removing that vertex from queue,whose neighbour will be visited now

v = Q.dequeue( ) // processing all the neighbours of v

for all neighbours w of v in Graph G

if w is not visited

Q.enqueue( w ) //Stores w in Q to further visit its neighbour

mark w as visited.

**Python code:**

class graph:

def \_\_init\_\_(self):

self.matrix=[]

def enter\_graph(self):

print("Enter No. of vertices and edges :")

n, m = input().strip().split(' ')

n, m = [int(n), int(m)]

for i in range(0,n):

self.matrix.append([])

for j in range(0,n):

self.matrix[i].append(0)

print("Enter Edges: ")

for a1 in range(m):

u, v = input().strip().split(' ')

u, v = [int(u), int(v)]

self.matrix[u-1][v-1] = 1

def bfs(self,s):

visited=[]

for i in range(0,len(self.matrix)):

visited.append(0)

queue=[]

queue.append(s)

visited[s-1]=1

while (len(queue)!=0) :

node=queue.pop(0)

print(node,sep=' ')

for x in range(0,len(visited)):

if(self.matrix[node-1][x]==1 and visited[x]==0):

visited[x]=1

queue.append(x+1)

g=graph()

g.enter\_graph()

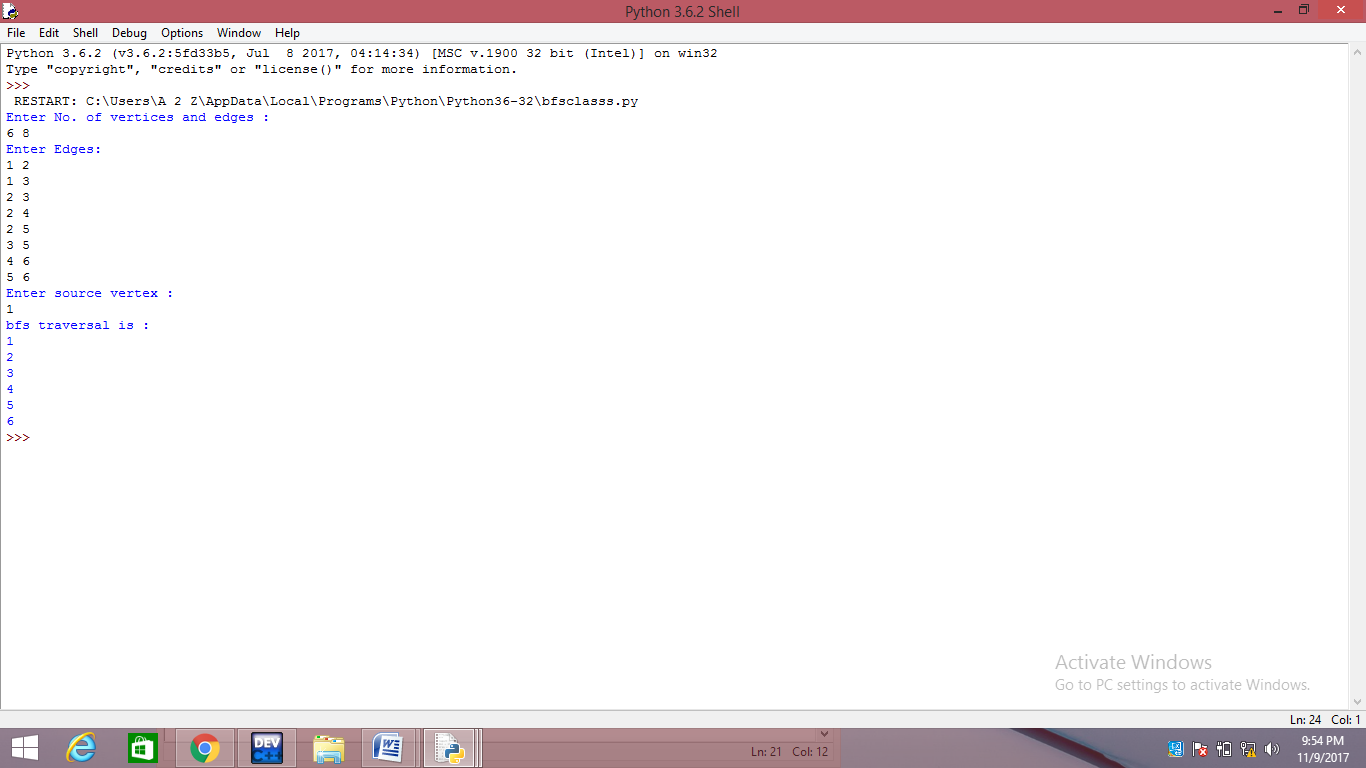
print("Enter source vertex :")

s=int(input())

print("bfs traversal is :")

g.bfs(s)

**Output:**

****

**2. DFS**

**Logic:**

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

This recursive nature of DFS can be implemented using stacks. The basic idea is as follows:  
Pick a starting node and push all its adjacent nodes into a stack.  
Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.  
Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

**Algorithm:**

DFS-recursive(G, s): // Where G is graph and s is the source vertex

mark s as visited

for all neighbours w of s in Graph G:

if w is not visited:

DFS-recursive(G, w)

**Python code :**

class graph:

def \_\_init\_\_(self):

self.matrix=[]

def enter\_graph(self):

print("Enter no. of vertices and Edges : ")

n, m = input().strip().split(' ')

n, m = [int(n), int(m)]

for i in range(0,n):

self.matrix.append([])

for j in range(0,n):

self.matrix[i].append(0)

print("Enter Edges : ")

for a1 in range(m):

u, v = input().strip().split(' ')

u, v = [int(u), int(v)]

self.matrix[u][v] = 1

self.matrix[v][u] = 1

def dfs(self,s,visited):

print(s)

for i in range(0,len(self.matrix)):

if(self.matrix[s][i] and visited[i]==0):

visited[i]=1

self.dfs(i,visited)

def dfs\_visit(self,s):

visited=[]

for i in range(0,len(self.matrix)):

visited.append(0)

visited[s]=1

self.dfs(s,visited)

g=graph()

g.enter\_graph()

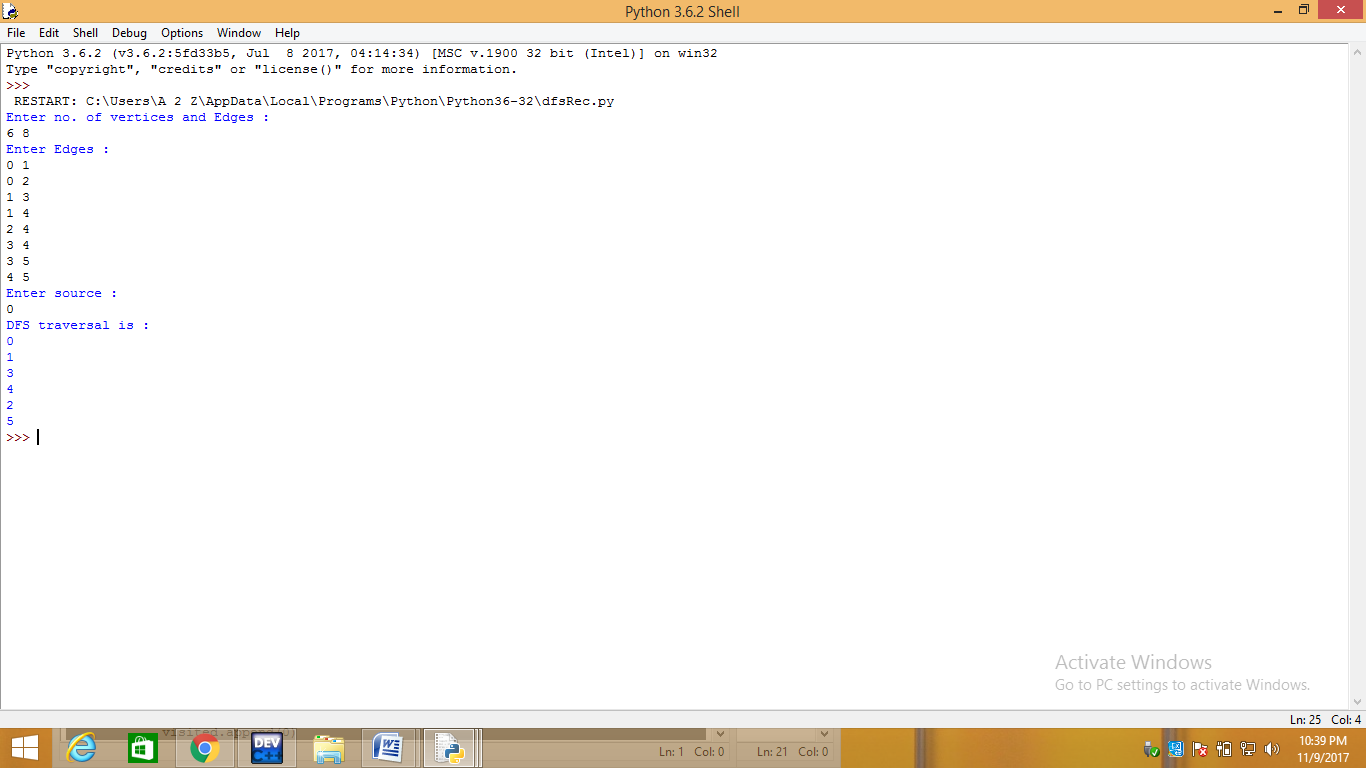
print("Enter source :")

s=int(input())

print("DFS traversal is :")

g.dfs\_visit(s)

**Output:**

****

1. **Dijkstra’s Shortest path**

**Logic :** Given a graph and a source vertex in graph, find shortest paths from source to all vertices in the given graph. we generate a SPT (shortest path tree) with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has minimum distance from source.

**Algorithm :**

Let the distance of start vertex from start vertex =0

Let the distance of all other vertices from start vertex= **INF**

Repeat

- Visit the unvisited vertex with the smallest known distance from the

start vertex.

- For the current vertex ,examine its unvisited neighbours .

- For the current vertex , calculate distance of each neighbor from the

Start vertex.

- If the calculated distance of a vertex is less than the known distance ,update

the shortest distance .

- Update the previous vertex for each of the updated distance .

- Add the current vertex to the list of visited vertices .

Until all vertices visited

**Python code :**

def minDistance(dist,flag,nv):

min=10000

for v in range(0,nv):

if(flag[v]==0 and dist[v]<=min):

min=dist[v]

ind=v

return ind

def printSolution(dist,nv):

print("vertex Distance from the source")

for i in range(0,nv):

print(i,dist[i],sep=" ")

def dijkstra(adj,nv,s):

dist=[]

flag=[]

for i in range(0,nv):

dist.append(10000)

flag.append(0)

dist[s]=0

for c in range(0,nv-1):

u=minDistance(dist,flag,nv)

flag[u]=1

for v in range(0,nv):

if(flag[v]==0 and adj[u][v] and dist[u]!=10000 and dist[u]+adj[u][v]<dist[v]):

dist[v]=dist[u]+ adj[u][v]

printSolution(dist,nv)

# main()

print("Enter no. of vertices and edges :")

nv, ne = input().strip().split(' ')

nv, ne = [int(nv), int(ne)]

adj=[]

for i in range(0,nv):

adj.append([])

for j in range(0,nv):

adj[i].append(0)

print("Enter edges with their weights :")

for a1 in range(1,ne+1):

u, v, w = input().strip().split(' ')

u, v, w = [int(u), int(v), int(w)]

adj[u][v] = w

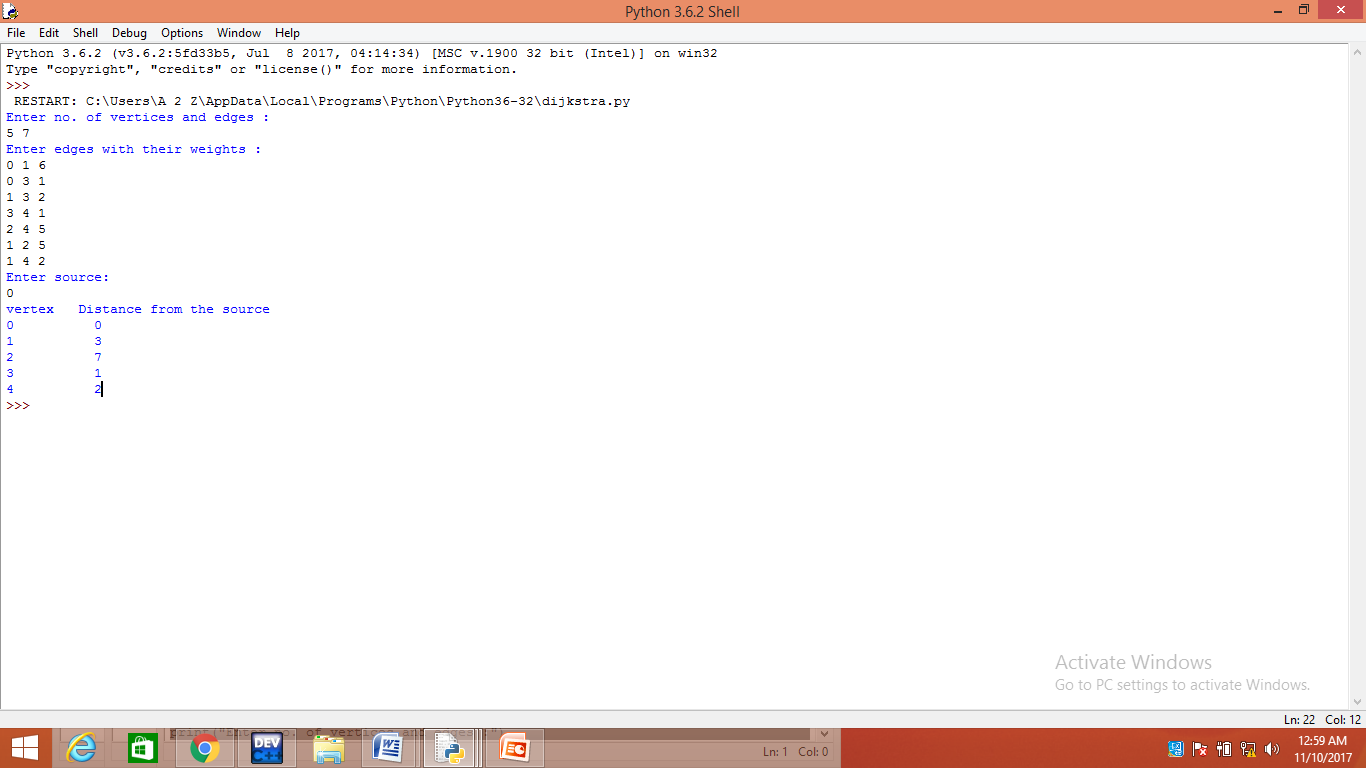
adj[v][u] = w

print("Enter source: ")

s=int(input())

dijkstra(adj,nv,s)

**Output**:



1. **Center of the graph**

**Logic :** The **center**  of a [graph](https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)) is the set of all vertices of minimum [eccentricity](https://en.wikipedia.org/wiki/Eccentricity_(graph_theory)),[[2]](https://en.wikipedia.org/wiki/Graph_center#cite_note-2) that is, the set of all vertices *A* where the greatest distance *d*(*A*,*B*) to other vertices *B* is minimal. Equivalently, it is the set of vertices with eccentricity equal to the graph's [radius](https://en.wikipedia.org/wiki/Radius_(graph_theory)).[[3]](https://en.wikipedia.org/wiki/Graph_center#cite_note-3)Thus vertices in the center (**central points**) minimize the maximal distance from other points in the graph.

**Algorithm :**

Let G is the Graph

1. For each vertex v in adjacency matrix

* Calculate minimum distance from v to other vertices and find the maximum of all distances and call it eccentricity of v .
* Vertices with minimum eccentricity are the center of the graph.

**Python code :**

def minDistance(dist,flag,nv):

min=1000000

for v in range(0,nv):

if(flag[v]==0 and dist[v]<=min):

min=dist[v]

ind=v

return ind

ecc=[]

def printSolution(dist,nv,s):

ecc.append(max(dist))

def dijkstra(adj,nv,s):

dist=[]

flag=[]

for i in range(0,nv):

dist.append(1000000)

flag.append(0)

dist[s]=0

for c in range(0,nv-1):

u=minDistance(dist,flag,nv)

flag[u]=1

for v in range(0,nv):

if(flag[v]==0 and adj[u][v] and dist[u]!=1000000 and dist[u]+adj[u][v]<dist[v]):

dist[v]=dist[u]+ adj[u][v]

printSolution(dist,nv,s)

# main()

print("Enter no. of vertices and Edges: ")

nv, ne = input().strip().split(' ')

nv, ne = [int(nv), int(ne)]

adj=[]

for i in range(0,nv):

adj.append([])

for j in range(0,nv):

adj[i].append(0)

print("Enter the Edges with their weight :")

for a1 in range(1,ne+1):

u, v, w = input().strip().split(' ')

u, v, w = [int(u), int(v), int(w)]

adj[u][v] = w

adj[v][u] = w

for i in range(0,nv):

dijkstra(adj,nv,i)

print("Center of the graphs are :")

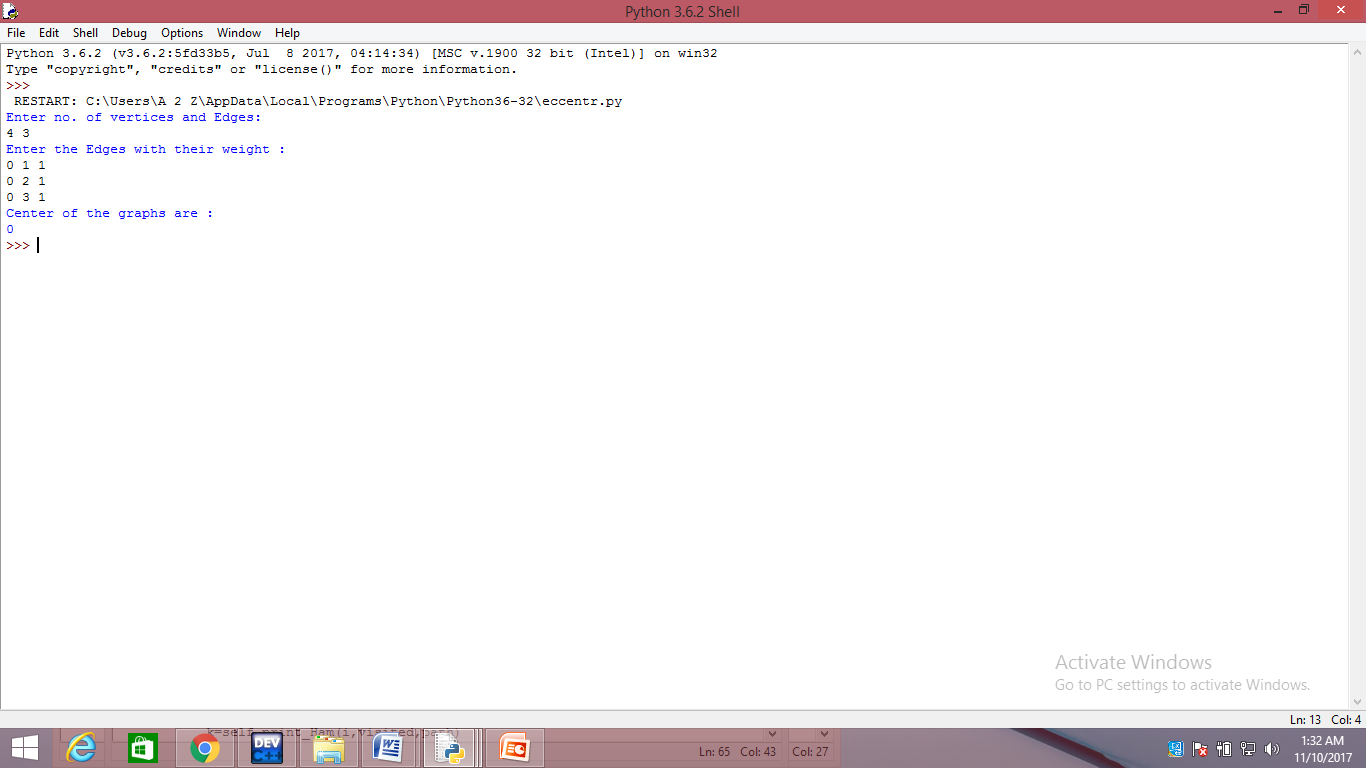
m=min(ecc)

for i in range(0,nv):

if(ecc[i]==m):

print(i,end=' ')

**Output:**

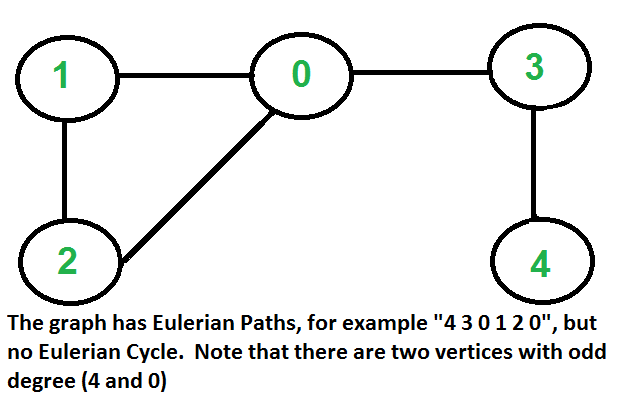


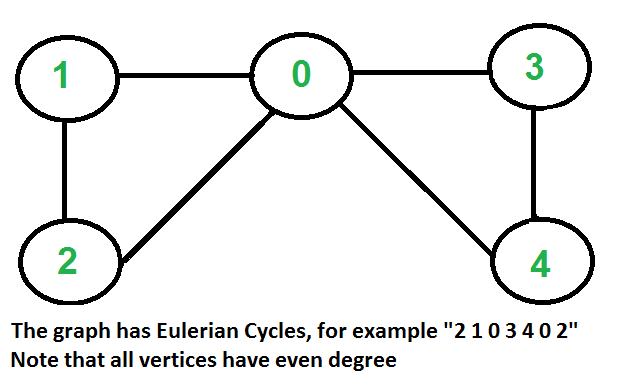
**EULER’s PATH**

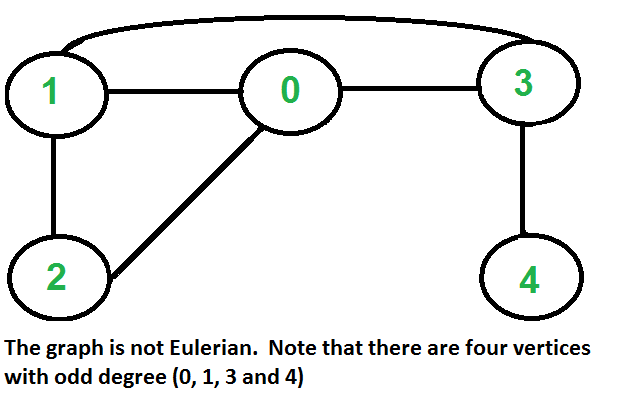
Eulerian path and circuit for undirected graph

[**3.4**](http://www.geeksforgeeks.org/medium/)

[Eulerian Path](http://en.wikipedia.org/wiki/Eulerian_path)is a path in graph that visits every edge exactly once. Eulerian Circuit is an Eulerian Path which starts and ends on the same vertex.

[](http://www.geeksforgeeks.org/wp-content/uploads/Euler1.png)

[](http://www.geeksforgeeks.org/wp-content/uploads/Euler2.png)

[](http://www.geeksforgeeks.org/wp-content/uploads/Euler3.png)

**LOGIC:**

**How to find whether a given graph is Eulerian or not?**  
The problem is same as following question. “Is it possible to draw a given graph without lifting pencil from the paper and without tracing any of the edges more than once”.

A graph is called Eulerian if it has an Eulerian Cycle and called Semi-Eulerian if it has an Eulerian Path. The problem seems similar to [Hamiltonian Path](http://www.geeksforgeeks.org/backtracking-set-7-hamiltonian-cycle/) which is NP complete problem for a general graph. Fortunately, we can find whether a given graph has a Eulerian Path or not in polynomial time. In fact, we can find it in O(V+E) time.

Following are some interesting properties of undirected graphs with an Eulerian path and cycle. We can use these properties to find whether a graph is Eulerian or not.

**Eulerian Cycle**  
An undirected graph has Eulerian cycle if following two conditions are true.

a) All vertices with non-zero degree are connected. We don’t care about vertices with zero degree because they don’t belong to Eulerian Cycle or Path (we only consider all edges).

b) All vertices have even degree.

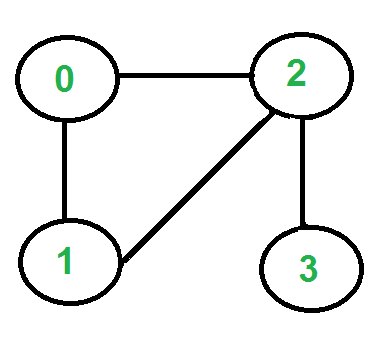
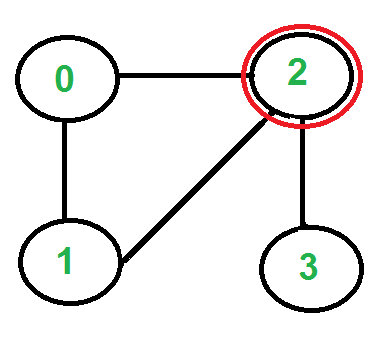
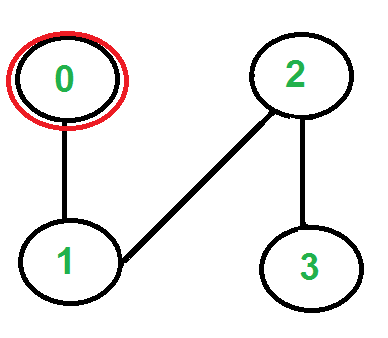
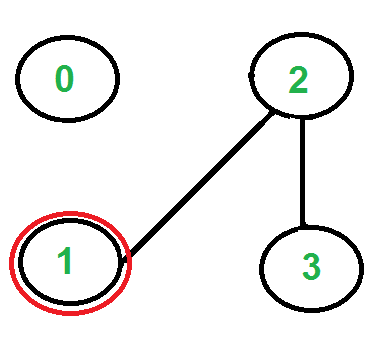
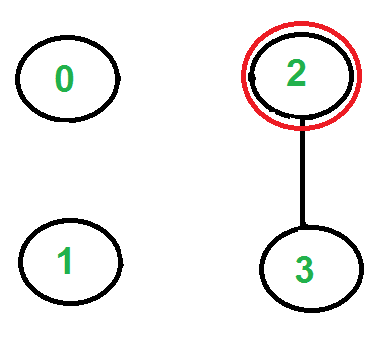
**Eulerian Path**   
An undirected graph has Eulerian Path if following two conditions are true.  
a) Same as condition (a) for Eulerian Cycle

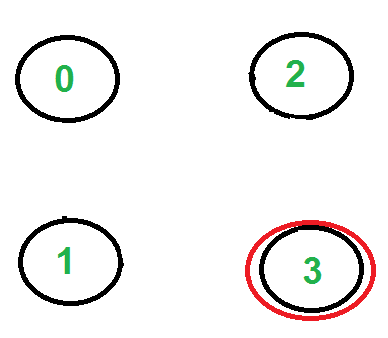
b) If zero or two vertices have odd degree and all other vertices have even degree. Note that only one vertex with odd degree is not possible in an undirected graph (sum of all degrees is always even in an undirected graph)

Note that a graph with no edges is considered Eulerian because there are no edges to traverse.

**How does this work?**  
In Eulerian path, each time we visit a vertex v, we walk through two unvisited edges with one end point as v. Therefore, all middle vertices in Eulerian Path must have even degree. For Eulerian Cycle, any vertex can be middle vertex, therefore all vertices must have even degree.

**ALGORITHM**

1. Enter the no. of vertex and the no. of edges.
2. Then enter the egdes.
3. Make an adjacency matrix.
4. Create an visited array and initialize for all i 1 to n eqaual to zero.
5. Following is Fleury’s Algorithm for printing Eulerian trail or cycle
6. Make sure the graph has either 0 or 2 odd vertices.
7. If there are 0 odd vertices, start anywhere. If there are 2 odd vertices, start at one of them.
8. Follow edges one at a time. If you have a choice between a bridge and a non-bridge, *always choose the non-bridge*.
9. Stop when you run out of edges.
10. The idea is, ***“don’t burn***[***bridges***](http://www.geeksforgeeks.org/bridge-in-a-graph/)***“*** so that we can come back to a vertex and traverse remaining edges. For example let us consider the following graph.
11. [](http://www.geeksforgeeks.org/wp-content/uploads/Euler11.png)
12. There are two vertices with odd degree, ‘2’ and ‘3’, we can start path from any of them. Let us start tour from vertex ‘2’.  
    [](http://www.geeksforgeeks.org/wp-content/uploads/Euler21.png)  
    There are three edges going out from vertex ‘2’, which one to pick? We don’t pick the edge ‘2-3’ because that is a bridge (we won’t be able to come back to ‘3’). We can pick any of the remaining two edge. Let us say we pick ‘2-0’. We remove this edge and move to vertex ‘0’.  
    [](http://www.geeksforgeeks.org/wp-content/uploads/Eule3.png)  
    There is only one edge from vertex ‘0’, so we pick it, remove it and move to vertex ‘1’. Euler tour becomes ‘2-0 0-1’.  
    [](http://www.geeksforgeeks.org/wp-content/uploads/Euler4.png)
13. There is only one edge from vertex ‘1’, so we pick it, remove it and move to vertex ‘2’. Euler tour becomes ‘2-0 0-1 1-2’  
    [](http://www.geeksforgeeks.org/wp-content/uploads/Euler5.png)

Again there is only one edge from vertex 2, so we pick it, remove it and move to vertex 3. Euler tour becomes ‘2-0 0-1 1-2 2-3’  
[](http://www.geeksforgeeks.org/wp-content/uploads/Euler6.png)

There are no more edges left, so we stop here. Final tour is ‘2-0 0-1 1-2 2-3’.

In the following code, it is assumed that the given graph has an Eulerian trail or Circuit. The main focus is to print an Eulerian trail or circuit. We can use [isEulerian()](http://www.geeksforgeeks.org/eulerian-path-and-circuit/" \t "_blank) to first check whether there is an Eulerian Trail or Circuit in the given graph.

We first find the starting point which must be an odd vertex (if there are odd vertices) and store it in variable ‘u’. If there are zero odd vertices, we start from vertex ‘0’. We call printEulerUtil() to print Euler tour starting with u. We traverse all adjacent vertices of u, if there is only one adjacent vertex, we immediately consider it. If there are more than one adjacent vertices, we consider an adjacent v only if edge u-v is not a bridge. How to find if a given is edge is bridge? We count number of vertices reachable from u. We remove edge u-v and again count number of reachable vertices from u. If number of reachable vertices are reduced, then edge u-v is a bridge. To count reachable vertices, we can either use BFS or DFS, we have used DFS in the above code. The function DFSCount(u) returns number of vertices reachable from u.  
Once an edge is processed (included in Euler tour), we remove it from the graph. To remove the edge, we replace the vertex entry with -1 in adjacency list. Note that simply deleting the node may not work as the code is recursive and a parent call may be in middle of adjacency list.

**CODE**

visited=[]

class graph:

def \_\_init\_\_(self):

self.matrix=[]

def enter\_graph(self):

print("\*\*Enter the vertices and the edges\*\*")

n, m = input().strip().split(' ')

n, m = [int(n), int(m)]

for i in range(0,n):

self.matrix.append([])

for j in range(0,n):

self.matrix[i].append(0)

# print(self.matrix)

print("Enter the edges")

for a1 in range(m):

u, v = input().strip().split(' ')

u, v = [int(u), int(v)]

self.matrix[u][v] = 1

self.matrix[v][u] = 1

def chooseOdd(self):

for i in range(0,len(self.matrix)):

ct=0

for j in range(0,len(self.matrix)):

if(self.matrix[i][j]==1):

ct=ct+1

if(ct%2!=0):

vert=i

break

self.Euler(vert)

def removeE(self,u,v):

self.matrix[u][v]=0

self.matrix[v][u]=0

def dfs(self,s,visited):

for i in range(0,len(self.matrix)):

if(self.matrix[s][i] and visited[i]==0):

visited[i]=1

self.dfs(i,visited)

def isconnected(self):

visited=[]

for i in range(0,len(self.matrix)):

visited.append(0)

visited[0]=1

self.dfs(0,visited)

for i in range(0,len(self.matrix)):

if(visited[i]==0):

return False

return True

def addEdge(self,u,v):

self.matrix[u][v]=1

self.matrix[v][u]=1

def Euler(self,vert):

stack=[]

stack.insert(0,vert)

while(len(stack)!=0):

ele=stack.pop()

flag=0

f=0

for i in range(0,len(self.matrix)):

if(self.matrix[ele][i]==1):

f=1

self.removeE(ele,i)

k=i

if(self.isconnected()):

print(ele,i,sep="-")

stack.insert(0,k)

flag=1

break

else:

self.addEdge(ele,k)

if(flag==0 and f==1):

self.removeE(ele,k)

print(ele,k,sep="-")

stack.insert(0,k)

def isEuler(self):

odd=0

for i in range(0,len(self.matrix)):

ct=0

for j in range(0,len(self.matrix)):

if(self.matrix[i][j]==1):

ct=ct+1

if(ct%2!=0):

odd=odd+1

if(odd==0):

g.Euler(0)

elif(odd==2):

g.chooseOdd()

else:

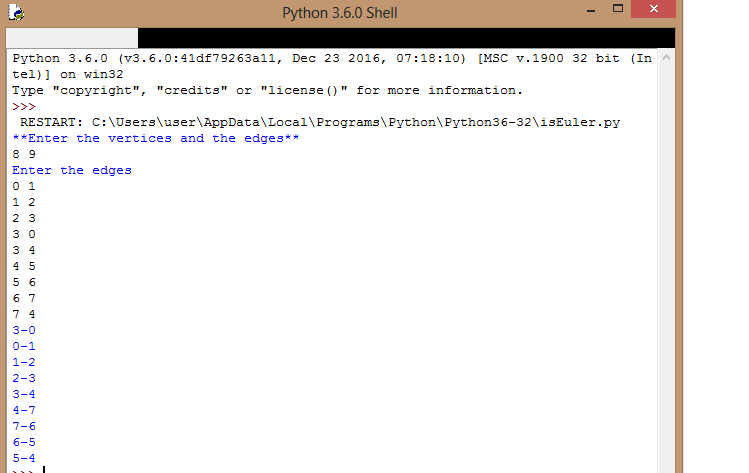
print("No Euler Path Or Circuit Exists")

g=graph()

g.enter\_graph()

g.isEuler()

**OUTPUT**

****

**HAMILTONIAN CYCLE**

Backtracking | Set 6 (Hamiltonian Cycle)

[**3.7**](http://www.geeksforgeeks.org/medium/)

[Hamiltonian Path](http://en.wikipedia.org/wiki/Hamiltonian_path) in an undirected graph is a path that visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in graph) from the last vertex to the first vertex of the Hamiltonian Path.Determine whether a given graph contains Hamiltonian Cycle or not. If it contains, then print the path. Following are the input and output of the required function.

*Input:*  
A 2D array graph[V][V] where V is the number of vertices in graph and graph[V][V] is adjacency matrix representation of the graph. A value graph[i][j] is 1 if there is a direct edge from i to j, otherwise graph[i][j] is 0.

*Output:*  
An array path[V] that should contain the Hamiltonian Path. path[i] should represent the ith vertex in the Hamiltonian Path. The code should also return false if there is no Hamiltonian Cycle in the graph.

For example, a Hamiltonian Cycle in the following graph is {0, 1, 2, 4, 3, 0}. There are more Hamiltonian Cycles in the graph like {0, 3, 4, 2, 1, 0}

(0)--(1)--(2)

| / \ |

| / \ |

| / \ |

(3)-------(4)

And the following graph doesn’t contain any Hamiltonian Cycle.

(0)--(1)--(2)

| / \ |

| / \ |

| / \ |

(3) (4)

**LOGIC:**  
Generate all possible configurations of vertices and print a configuration that satisfies the given constraints. There will be n! (n factorial) configurations.

while there are untried conflagrations

{

generate the next configuration

if ( there are edges between two consecutive vertices of this

configuration and there is an edge from the last vertex to

the first ).

{

print this configuration;

break;

}

}

**Backtracking Algorithm**  
Create an empty path array and add vertex 0 to it. Add other vertices, starting from the vertex 1. Before adding a vertex, check for whether it is adjacent to the previously added vertex and not already added. If we find such a vertex, we add the vertex as part of the solution. If we do not find a vertex then we return false.

**ALGORITHM**

STEPS:

1. Enter the no. of vertex and the no. of edges.
2. Then enter the egdes.
3. Make an adjacency matrix.
4. Create an visited array and initialize for all i 1 to n eqaual to zero.
5. Create a path array(empty stack).
6. For each vertex v in adjacency matrix vertex call the Hamiltonian path function(v).

Hamiltonian path of (v):

If (len(path)==no. of vertex)

Print (path);

Return true;

For each vertex v

Check adjacent vertex w of v

If visited[w]==false

Visited[w]=1;

Path.push(w);

If(Hamiltonian path(w)==true)

Return true;

Visited[w]=0;

Path.pop();

1. End.

**CODES**

class graph:

def \_\_init\_\_(self):

self.matrix=[]

def enter\_graph(self):

print("\*\*Enter no of vertices and edges\*\*")

n, m = input().strip().split(' ')

n, m = [int(n), int(m)]

for i in range(0,n):

self.matrix.append([])

for j in range(0,n):

self.matrix[i].append(0)

print("Enter the edges")

for a1 in range(m):

u, v = input().strip().split(' ')

u, v = [int(u), int(v)]

self.matrix[u][v] = 1

self.matrix[v][u] = 1

def print\_Ham(self,v,visited,path):

N=len(self.matrix)

if(len(path)==N):

print(path,end=' ')

print("\n")

return True

for i in range(0,N):

if(self.matrix[v][i]==1 and visited[i]==0):

visited[i]=1

path.insert(0,i)

if(self.print\_Ham(i,visited,path)==True):

return True

#backtrack

visited[i]=0

path.pop(0)

return False

def print\_visit(self):

visited=[]

for i in range(0,len(self.matrix)):

visited.append(0)

path=[]

for i in range(0,len(self.matrix)):

for j in range(0,len(self.matrix)):

visited[j]=0

if(len(path)!=0):

path.pop(0)

path.append(i)

visited[i]=1

k=self.print\_Ham(i,visited,path)

if(k==True):

return True

return False

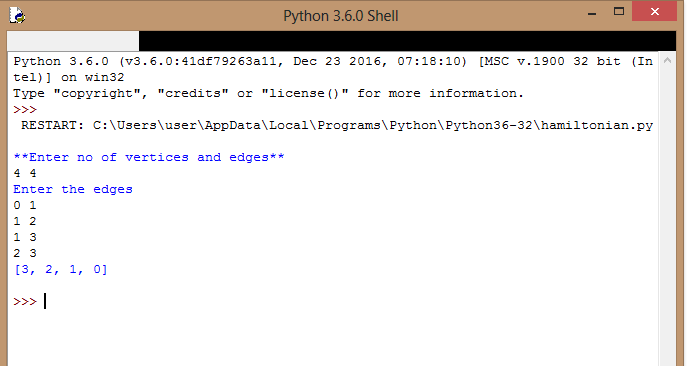
g=graph()

g.enter\_graph()

if(g.print\_visit()==False):

print("no Hamiltnonian Path exists")

**OUTPUT:**

****

**TRAVELLING SALES MAN PROBLEM**

**Travelling Salesman Problem (TSP):** Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.  
Hamiltoninan cycle problem is to find if there exist a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.

For example, consider the graph shown in figure on right side. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is 10+25+30+15 which is 80.

The problem is a famous [NP hard](http://www.geeksforgeeks.org/np-completeness-set-1/)problem. There is no polynomial time know solution for this problem.

Following are different solutions for the traveling salesman problem.

**LOGIC:**

1) Consider city 1 as the starting and ending point.  
2) Generate all (n-1)! [Permutations](http://www.geeksforgeeks.org/write-a-c-program-to-print-all-permutations-of-a-given-string/)of cities.  
3) Calculate cost of every permutation and keep track of minimum cost permutation.  
4) Return the permutation with minimum cost.

**ALGORITHM**

1. Enter the no. of vertex and the no. of edges.
2. Then enter the egdes.
3. Make an adjacency matrix.
4. Create an visited array and initialize for all i 1 to n eqaual to zero.
5. Initialise the visited array with zero.
6. Start from ay vertex u.
7. Call TSP\_min.cost(u)

Visit[u]=1;

Print(u)

8.Find nearest vertex from u and add cost to min cost and now recursively call the TSP\_min.cost.

9.End.

**CODE**

class graph:

def \_\_init\_\_(self):

self.matrix=[]

self.cost=0

def enter\_graph(self):

print("\*\*Enter the vertices and edges\*\*")

n, m = input().strip().split(' ')

n, m = [int(n), int(m)]

for i in range(0,n):

self.matrix.append([])

for j in range(0,n):

self.matrix[i].append(0)

#print(self.matrix)

print("Enter the edges with respective weights")

for a1 in range(m):

u, v, w= input().strip().split(' ')

u, v, w = [int(u), int(v) ,int(w)]

self.matrix[u][v] = w

self.matrix[v][u] = w

print(self.matrix)

def tsp(self,c,visit):

minm=999

nearest\_city=999

for i in range(0,len(self.matrix)):

if(self.matrix[c][i]!=0 and visit[i]==0):

if(self.matrix[c][i] <minm):

minm= self.matrix[c][i]

temp = self.matrix[c][i];

nearest\_city = i

if(minm != 999):

self.cost = self.cost + temp

return nearest\_city

def min\_cost(self,city,visit):

visit[city] = 1;

print(city,end=" ")

nearest\_city = self.tsp(city,visit)

if(nearest\_city==999):

nearest\_city=0

print(nearest\_city)

self.cost = self.cost + self.matrix[city][nearest\_city]

return

self.min\_cost(nearest\_city,visit);

def visit(self):

visit=[]

for i in range(0,len(self.matrix)):

visit.append(0)

self.min\_cost(0,visit)

print(self.cost)

#main()

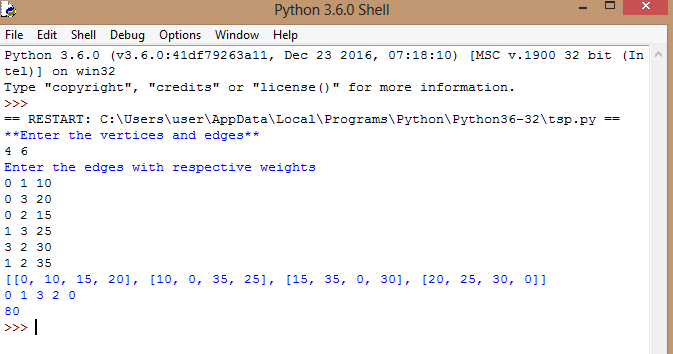
g=graph()

g.enter\_graph()

visited=[]

g.visit()

**OUTPUT:**

****

**RELATION MATRIX**

The problem is based on whether a matrix relation is reflexive, irreflexive, symmetric, anti-symmetric, asymmetric, transitive or not.

**LOGIC:**

If an element has a relation with itself is called reflexive. If R is a relation on set A, then A is reflexive if (a, a) belongs to R for all a belongs to A.

If any of the element is not related with itself then the relation is called irreflexive. If R is a relation on set A, then A is said to be irreflexive if (a, a) does not belongs to R for all a belongs to A.

If relation R on set A is said to be symmetric if (a, b) belongs to R, that is aRb present then bRa must be present.

A relation R on a set A is asymmetric if whenever aRb, but a is not related to b

A relation R on a set A is anti-symmetric when aRb and bRa then a=b. contrapositive: A relation R on a set A is contrapositive , whenever a !=b then a is not related b or b is not related a.

A relation R on set A is transitive if (a, b) belongs to R, (b, c) belongs to R then (a, c) belongs to R.

**ALGORITHM:**

STEPS:

1. Enter the no. of vertex (say v) or no. of element in the set.
2. Then enter the a[v][v] adjacency matrix.
3. If a[i][j]==1 for all i==j then relation is reflexive.
4. If for any i==j a[i][j]!= 1 then its is irreflexive.
5. If a[i][j]==1 then check a[j][i] must be 1 then relation is said to be symmetric
6. Else it is Asymmetric.
7. If a relation is irreflexive and asymmetric then it must be Anti-symmetric.
8. If a[i][j]==1, check for any k, a[j][k]==1 or not
9. If a[j][k]==1 then check a[k][i]==1 or not,
10. If for any k a[k][i]!=1 then make flag=1, which was initialize with 0 and break, relation is not transitive.

**CODE**

/\*C program to implement relation for any graph \*/

#include<stdio.h>

#include<stdlib.h>

#define max 10

int main()

{

int a[max][max],v,i,j,c=0,c1=0,p=0,k,flag=0;

char chr;

printf("Enter the number of vertex : ");

scanf("%d",&v);

printf("enter adjacency matrix\n");

for(i=0;i<v;i++)

{

for(j=0;j<v;j++)

scanf("%d",&a[i][j]);

printf("\n");

}

printf("The matrix is :-\n");

for(i=0;i<v;i++)

{

for(j=0;j<v;j++)

{

printf("%d ",a[i][j]);

}

printf("\n");

}

p=((v-1)\*v);

for(i=0;i<v;i++)

{

for(j=0;j<v;j++)

{

if(i==j)

{

if(a[i][j]==1)

c++;

}

else

{

if(a[i][j]==a[j][i])

c1++;

}

if(a[i][j]==1)

{

for(k=0;k<v;k++)

{

if(a[j][k]==1)

{

if(a[k][i]!=1)

{

flag=1;

break;

}

}

}

}

}

}

if(c==v)

printf("\nRelation is Reflexive");

else

printf("\nRelation is not Reflexive");

if(c==0)

printf("\nRelation is Irreflexive");

else

printf("\nRelation is not Irreflexive");

if(c1==p)

printf("\nRelation is Symmetric");

else

printf("\nRelation is Asymmetric");

if(flag==0)

printf("\nRelation is Transitive");

else

printf("\nRelation is not Transitive");

if(c==0 && c1!=p)

printf("\nRelation is Antisymmetric");

else

printf("\nRelation is Antisymmetric");

return 0;

}

OUPTUT

