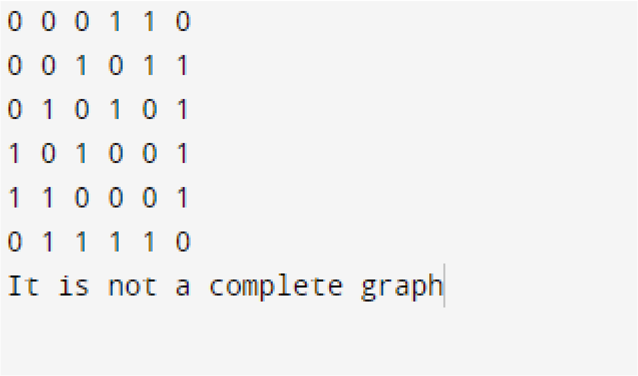
Practical I

Q1. WAP to represent Graphs using the Adjacency matrix and check if it is a complete graph.

```
#include<iostream>
using namespace std;
int vertArr[20][20]; //the adjacency matrix initially 0
int count = 0;
void displayMatrix(int v) {
  int i, j;
  for (i = 0; i < v; i++) {
    for (j = 0; j < v; j++) {
      cout << vertArr[i][j] << " ";
      if(vertArr[i][j]==1)
        count++;
    }
    cout << endl;
  }
}
void add_edge(int u, int v) { //function to add edge into the matrix
  vertArr[u][v] = 1;
  vertArr[v][u] = 1;
}
main(int argc, char * argv[]) {
  int v = 6; //there are 6 vertices in the graph
  int edge = v*(v-1)/2;
  add_edge(0, 4);
  add_edge(0, 3);
  add_edge(1, 2);
  add_edge(1, 4);
  add_edge(1, 5);
  add_edge(2, 3);
  add_edge(2, 5);
  add_edge(5, 3);
  add_edge(5, 4);
  displayMatrix(v);
  if(count==edge)
    cout<<"It is a complete graph";
  else
    cout<<"It is not a complete graph";
}
```

Output:



```
0 0 0 1 1 0
0 0 1 0 1 1
0 1 0 1 0 1
1 0 1 0 0 1
1 1 0 0 0 1
0 1 1 1 1 0
It is not a complete graph
```

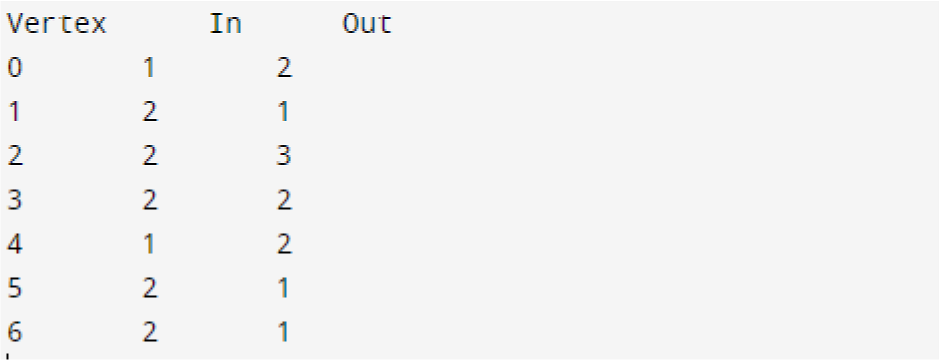
Q 2. WAP to accept a directed graph and compute in degree and out degree of each vertex.

```
#include <bits/stdc++.h>
using namespace std; // Function to print the in and out degrees
// of all the vertices of the given graph
void findInOutDegree(list < list < int >> adjlist,
  int n) {
  int * iN = new int[n]();
  int * ouT = new int[n]();
  list < list < int > > ::iterator nest_list;
  int i = 0;
  for (nest_list = adjlist.begin(); nest_list != adjlist.end(); nest_list++) {
    list < int > lst = * nest_list;
    // Out degree for ith vertex will be the count
    // of direct paths from i to other vertices
    ouT[i] = lst.size();
    for (auto it = lst.begin(); it != lst.end(); it++) {
      // Every vertex that has an incoming
      // edge from i
      iN[ * it]++;
    }
    i++;
  }
  cout << "Vertex\t\tIn\t\tOut" << endl;
  for (int k = 0; k < n; k++) {
    cout << k << "\t\t" <<
      iN[k] << "\t\t" <<
      ouT[k] << endl;
  }
}
```

```
// Driver code
int main() {
   // Adjacency list representation of the graph
   list < list < int >> adjlist;
   // Vertices 1 and 2 have an incoming edge
   // from vertex 0
   list < int > tmp;
   tmp.push_back(1);
   tmp.push_back(2);
   adjlist.push_back(tmp);
   tmp.clear();
   // Vertex 3 has an incoming edge
   // from vertex 1
   tmp.push_back(3);
   adjlist.push_back(tmp);
   tmp.clear();
   // Vertices 0, 5 and 6 have an incoming
   // edge from vertex 2
   tmp.push_back(0);
   tmp.push_back(5);
   tmp.push_back(6);
   adjlist.push_back(tmp);
   tmp.clear();
   // Vertices 1 and 4 have an incoming
   // edge from vertex 3
   tmp.push_back(1);
   tmp.push_back(4);
   adjlist.push_back(tmp);
   tmp.clear();
   // Vertices 2 and 3 have an incoming
   // edge from vertex 4
   tmp.push_back(2);
   tmp.push_back(3);
   adjlist.push_back(tmp);
   tmp.clear();
   // Vertices 4 and 6 have an incoming
   // edge from vertex 5tmp.push_back(4);
   tmp.push_back(6);
   adjlist.push_back(tmp);
   tmp.clear();
   // Vertex 5 has an incoming
   // edge from vertex 6
   tmp.push_back(5);
   adjlist.push_back(tmp);
```

```
    tmp.clear();
    int n = adjlist.size();
    findInOutDegree(adjlist, n);
}
```

Output:

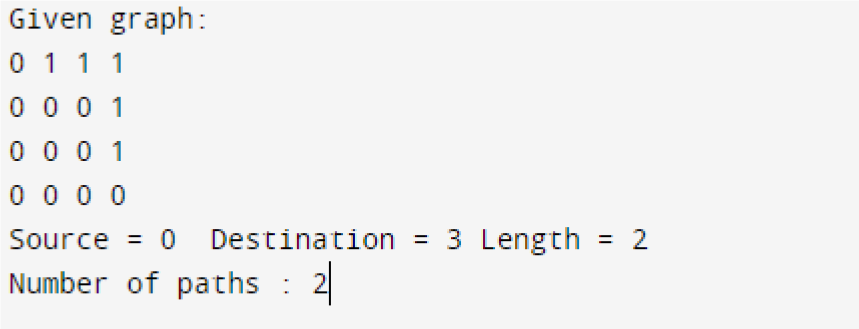


```
Vertex       In       Out
0        1        2
1        2        1
2        2        3
3        2        2
4        1        2
5        2        1
6        2        1
```

Q 3. Given a graph, WAP to find the number of paths of length n between source and destination entered by user.

```
#include <iostream>
using namespace std;
#define V 4
// A naive recursive function to count
// walks from u to v with k edges
int countwalks(int graph[][V], int u, int v, int k)
{
    if (k == 0 && u == v)
        return 1;
    if (k == 1 && graph[u][v])
        return 1;
    if (k <= 0)
        return 0;
    int count = 0;
    for (int i = 0; i < V; i++)
        if (graph[u][i] == 1) // Check if is adjacent of u
            count += countwalks(graph, i, v, k - 1);
    return count;
}
void displayMatrix() {

}
int main()
{
    int graph[V][V] = { { 0, 1, 1, 1 },
                        { 0, 0, 0, 1 },
                        { 0, 0, 0, 1 },
                        { 0, 0, 0, 0 } };
    int u = 0, v = 3, n = 2;
    cout<<"Given graph: "<<endl;
    for (int i = 0; i < V; i++) {
      for (int j = 0; j < V; j++) {
        cout << graph[i][j] << " ";
       }
      cout << endl;
    }
    cout<<"Source = " << u<<"\tDestination = "<<v<<"\tLength = "<<n<<endl;
    cout << "Number of paths : " <<countwalks(graph, u, v, n);
    return 0;
}
```

Output:

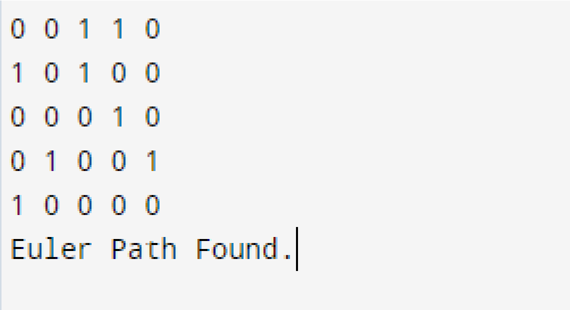


```
Given graph:
0 1 1 1
0 0 0 1
0 0 0 1
0 0 0 0
Source = 0  Destination = 3 Length = 2
Number of paths : 2
```

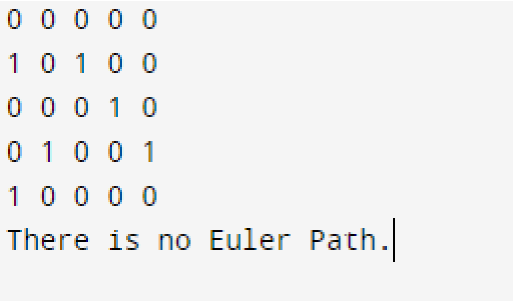
Q4. Given an adjacency matrix of a graph, write a program to check whether a given set of vertices forms an Euler path.

```
#include<iostream>
#include<vector>
#define NODE 5
using namespace std;
int graph[NODE][NODE] = {
  {0,0,0,0,0},
  {1,0,1,0,0},
  {0,0,0,1,0},
  {0,1,0,0,1},
  {1,0,0,0,0}};
void traverse(int u, bool visited[]) {
  visited[u] = true; //mark v as visited
  for (int v = 0; v < NODE; v++) {
    if (graph[u][v]) {
      if (!visited[v])
        traverse(v, visited);
    }
  }
}
bool isConnected() {
  bool * vis = new bool[NODE];
  //for all vertex u as start point, check whether all nodes are visible or not
  for (int u; u < NODE; u++) {
    for (int i = 0; i < NODE; i++)
      vis[i] = false; //initialize as no node is visited
    traverse(u, vis);
    for (int i = 0; i < NODE; i++) {
      if (!vis[i]) //if there is a node, not visited by traversal, graph is not connected
        return false;
    }
  }
  return true;
}

void displayMatrix() {
  int i, j;
  for (i = 0; i < NODE; i++) {
    for (j = 0; j < NODE; j++) {
      cout << graph[i][j] << " ";
    }
    cout << endl;
  }
}


bool hasEulerPath() {
  int an = 0, bn = 0;
  if (isConnected() == false) { //when graph is not connected
    return false;
  }
  vector < int > inward(NODE, 0), outward(NODE, 0);
  for (int i = 0; i < NODE; i++) {
    int sum = 0;
    for (int j = 0; j < NODE; j++) {
      if (graph[i][j]) {
        inward[j]++; //increase inward edge for destination vertex
        sum++; //how many outward edge
      }
    }
    outward[i] = sum;
  }
  //check the condition for Euler paths
  if (inward == outward) //when number inward edges and outward edges for each node is same
    return true; //Euler Circuit, it has Euler path
  for (int i = 0; i < NODE; i++) {
    if (inward[i] != outward[i]) {
      if ((inward[i] + 1 == outward[i])) {
        an++;
      } else if ((inward[i] == outward[i] + 1)) {
        bn++;
      }
    }
  }
  if (an == 1 && bn == 1) { //if there is only an, and bn, then this has euler path
    return true;
  }
  return false;
}
int main() {
    displayMatrix();
  if (hasEulerPath())
    cout << "Euler Path Found.";
  else
    cout << "There is no Euler Path.";
}
```

Output:



```
0 0 1 1 0
1 0 1 0 0
0 0 0 1 0
0 1 0 0 1
1 0 0 0 0
Euler Path Found.
```

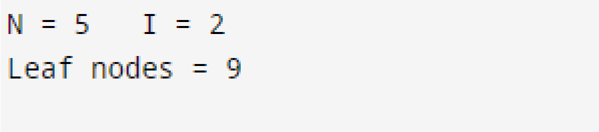


```
0 0 0 0 0
1 0 1 0 0
0 0 0 1 0
0 1 0 0 1
1 0 0 0 0
There is no Euler Path.
```

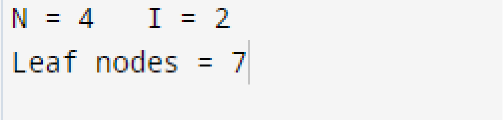
Q5. Given a full n-arry tree with I internal vertices, WAP to find number of leaf nodes.

```
#include <bits/stdc++.h>
using namespace std;
// Function to calculate
// leaf nodes in n-ary tree
int calcNodes(int N, int I) {
  int result = 0;
  result = I * (N - 1) + 1;
  return result;
}
// Driver code
int main() {
  int N = 5, I = 2;
  cout << "Leaf nodes = " << calcNodes(N, I);
  return 0;
}
```

Output:



```
N = 5   I = 2
Leaf nodes = 9
```



```
N = 4   I = 2
Leaf nodes = 7
```