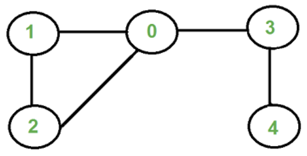
CSCI-311 **Project 4** **Graphs**

In this project you will implement an undirected graph that is represented as an Adjacency Lists (implemented as vector<vector< edge > > Adj). You will use *struct **edge*** that contains two data members: an integer *neighbor* and an integer *w.*

Input file:

```
5 5
0 1
0 2
0 3
1 2
3 4
```

Undirected graph G:



Adjacency lists *Adj*

0	1	2	3
1	0	2	
2	0	1	
3	0	4	
4	3		

Step 1. Implement a class Ugraph (stands for *undirected graph*). You need to write two files: *ugraph.h* and *ugraph.cpp.*

Inside *ugraph.h*, declare and define *struct **edge***:

```
struct edge{
        int neighbor; // adjacent node
        int w; //keeps auxiliary information
        edge(){
                neighbor = 0;
                w = 0;
        };
        edge(int i, int j){
                neighbor = i;
                w = j;
        };
};
```

Class *Ugraph* will have the following data members and constructors (and other member functions):

```
//private data members
vector<vector<edge> > Adj;
vector<int> parents;
vector<int> distance;
vector<char> colors;
vector<TimeStamp> stamps;
//public constructor and member functions
Ugraph(int N);
void addEdge(int u, int v);
void removeEdge(int u, int v);
```

Constructor. The constructor of the class takes a single parameter, an integer N, and resizes *Adj, parents, stamps, distance* and *colors* to the size N. Initialize *parents* array: parent of a vertex v is equal to v. Initialize *distance* array with INT_MAX. Initialize *colors* with 'W' (stands for White).

void addEdge(int u, int v): This member function inserts a new edge between the existing vertices u and v in the graph. Since this is an undirected graph, this function must add *edge*(v, 0) to Adj[u] and add *edge*(u, 0) to Adj[v], where 0 is used to initialize *w* data member of an *edge*.

void removeEdge(int u, int v): This function will remove edge (u, v) from a graph. Since this is an undirected graph, two Adjacency lists must be modified: one of u and another of v. Here is how the removal must be implemented:
1) In Adj[u], find an *edge* with *neighbor* equal to v, assume that it is found at index *j.*
2) Copy the last *edge* in Adj[u] into *j-th* entry of Adj[u] (hence, *edge* with v becomes overwritten).
3) Resize Adj[u] to (Adj[u].size() - 1) size.
Repeat the same steps for Adj[v] and *edge* with *neighbor* equal to u.

BFS and DFS. You will need to implement BFS and DFS member functions.

printGraph. Write a public member function ***printGraph*** that prints Adjacency lists in the order: all nodes in the Adj[0], then all nodes in the Adj[1], and so on. This function will not print out *w* of each *edge,* only *neighbor.* Print out ***endl*** after each Adjacency list, and a space after each node (neighbor).

Step 2. Solve the following problems using Ugraph. You may add any member functions to class Ugraph to solve these problems. You may use *w* of struct *edge* to hold any information at each edge to help you to solve a problem. Your member functions may NOT change *Adj* (you may modify *w* at each edge, but you cannot remove or add edges of *Adj*), i.e. if you need to modify *Adj*, simply make a copy of *Adj* and pass this copy by reference to your member functions.

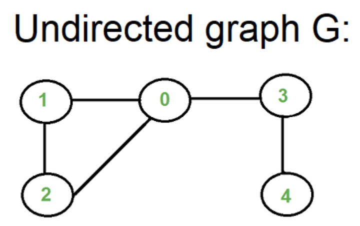
Important: All of these problems must be solved in O(V + E)-time unless stated otherwise. In other words if you need a few functions to accomplish a task (to solve a problem), then all of these functions must run in O(V + E)-time.

Problem 1. Given an undirected graph G and two vertices *u* and *v* of G, design an algorithm that will return *true* if there are two distinct paths from *u* to *v* in G and will print these two paths; otherwise, it will return *false* and will not print anything.

Your functions must run in O(V + E)-time.

Definition: two paths in an undirected graph G are ***distinct*** if they do not share any edges.

For example, given an undirected graph G in Figure below, and two vertices u = 0 and v = 2, there are two distinct paths from 0 to 2: (1) 0, 1, 2 and (2) 0, 2. So your program will print them in this format:



<node><space><node><space>…<endl>
Each path will be printed out on a separate line. The output for this example will be this:

```
0 1 2
0 2
```

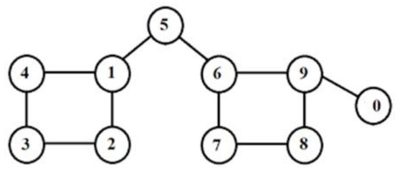
However, if we give two other vertices u = 1 and v = 3, then two paths (1) 1, 2, 0, 3 and (2) 1, 0, 3 are not distinct because they share edge (0, 3). So, there are no two distinct paths in this graph for vertices 1 and 3. Your program will return false and will not print anything.

You will need to write the following public member function (with exact name, parameters, return type):
bool ***distinctPaths***(int u, int v).
You may write any other member functions to solve this problem, but these functions must be called from *distinctPaths*.

Problem 2. Given an undirected connected graph G, find and print all ***bridge*** edges of G.

Your functions must run in O(V + E)-time.

Definition: in undirected connected graph G, an edge is called ***bridge*** if removal of this edge will disconnect G.



For example, in the graph on Figure to the left, edge (1, 5) is a bridge edge because if we remove this edge from the graph, the graph becomes not connected. Edge (6, 9) is not a bridge edge, because if is removed from the graph, the graph will remain connected.

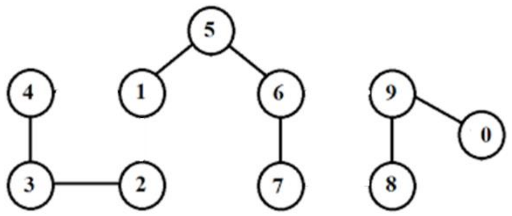
You will need to write the following public member function:
void ***printBridges***()
You may write any member functions in addition to *printBridges* to solve this problem, but these functions must be called from *printBridges*.

Output format: as soon as an edge (u, v) has been identified as a bridge, print it in the format
<u><space><v><endl>

Problem 3. Given an undirected graph G, find all connected components of G and print out the vertices in each component on a separate line. The order of printing is the following: all vertices in Connected Component with ID 0 on the first line, nodes in Connected Component with ID 1 on the second line, and so on.

Your functions must run in O(V + E)-time.

Definition: A **connected component** of an undirected graph G is a subgraph H of G such that H is connected (i.e. any two vertices in H are connected via a path in H).



For example, on given the graph on Figure to the left, your program will print out:

```
0 8 9
1 5 6 7
2 3 4
```

You need to write the following public member function:
void ***printCC***()

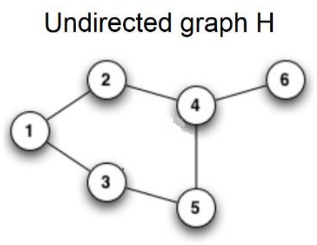
You may write any member functions in addition to this function, but they must be called from *printCC*.
Output format of a single line:
<node><space>…<node><space><endl>

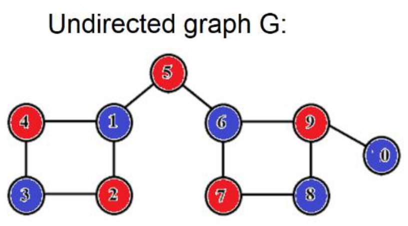
Problem 4. Given an undirected connected graph G, design an algorithm that returns *true* if it is possible to color all vertices of G in two colors, Red and Blue, such that if there is an edge (u, v) in G, then u and v must be colored with different colors.

To solve this problem, time complexity is <u>**not**</u> restricted to O(V + E)-time. Your functions may use O(VE)-time.

Note that if an undirected connected graph has an odd-length cycle, then this coloring is impossible. So, your program needs to check if there is an odd-length cycle in the graph, and if so, then it will return *false*. Otherwise, it will return *true.*

For example graph G below does not have an odd-length cycle, so it is possible to color G using two colors so that the adjacent vertices are colored with different colors, but graph H has an odd cycle, so this coloring is impossible.





Your program must contain the following member function:
bool ***twoColoring***()

You may write any other member functions to accomplish this task, but they must be called from *twoColoring.*

Grading: This project is worth 400 points. If your program does not compile, you will receive 0. Grading will be done according to the number of solved problems: each problem is worth 100 points. Each problem will be graded according to the passed tests on turnin. For example, if your program passed t11, but did not pass t12, then for the problem “printCC”, you will receive 100/2 = 50pts.

Since the code for addEdge, removeEdge, bfs and dfs would be given to you during lecture time, you will not receive any partial credit for these functions.

Submission: Submit *ugraph.h* and *ugraph.cpp* to turnin.

To test your program, use:

```
./run < tests/t01.in > t01.my
diff t01.my tests/t01.out
```

Member function	**Test files**
addEdge, printGraph	t01, t02
bfs	t02
removeEdge	t03, t04, t05
dfs	t05, t06
distinctPaths	t07, t08
printBridges	t09, t10
printCC	t11, t12
twoColoring	t13, 14, t15, t16