Walchand College of Engineering, Sangli
Computer Science & Engineering
Third Year

Course: Design and analysis of algorithm Lab

Lab course coordinator:
Dr. B. F. Momin- Batch: - T6, T7, T8
Mr. Kiran P. Kamble- Batch: - T1, T2, T3, T4, T5

Week 10 Assignment

Name: Trupti Rajendra Patil

Prn : 2020BTECS00051

Back Tracking

1) Implement the following using Back Tracking
   a) 4-Queen's problem

      Algorithm:

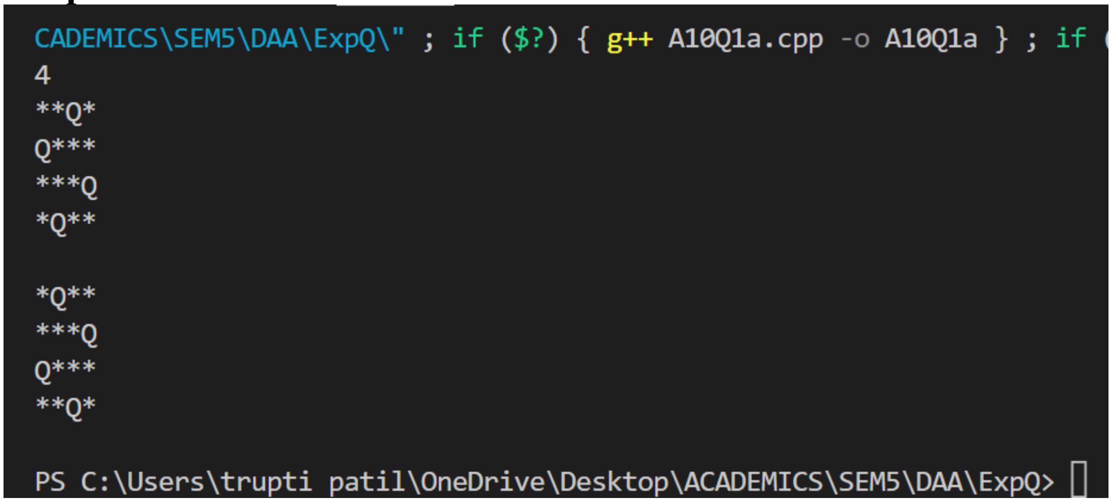
      1. Iterate through every row and check if it is safe to place the queen at that position
      2. To check if it is safe or not, check the upper left diagonal left row to that cell and lower left diagonal.
      3. If it is safe to place the queen then marks it else backtrack.
      4. Repeat this process for all rows.

      Code:

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

// Complexity O(n^2)

bool isSafe(int row,int col,vector<string> board,int n){
    int duprow=row;
    int dupcol=col;

    while(row>=0 && col>=0){
        if(board[row][col]=='Q'){
            return false;
        }
        row--;
        col--;
```

```cpp
    }

    row=duprow;
    col=dupcol;

    while(col>=0){
        if(board[row][col]=='Q'){
            return false;
        }
        col--;
    }

    row=duprow;
    col=dupcol;
    while(col>=0 && row<n){
        if(board[row][col]=='Q'){
            return false;
        }
        row++;
        col--;
    }

    return true;
}

void solve(int col,vector<string> &board,vector<vector<string>>
&ans,int q){
    if(col==q){
        ans.push_back(board);
        return;
    }

    for(int row=0;row<q;row++){
        if(isSafe(row,col,board,q)){
            board[row][col]='Q';
            solve(col+1,board,ans,q);
            board[row][col]='*';
        }
    }
}

int main(){
    int q;
    cin>>q;

    vector<vector<string>> ans;
    vector<string> board(q);
    string s(q,'*');
    for(int i=0;i<q;i++){
        board[i]=s;
    }

    solve(0,board,ans,q);

    for(int i=0;i<q;i++){
        for(int j=0;j<q;j++){
            cout<<ans[i][j]<<endl;
        }cout<<endl;
    }cout<<endl;
}
```

Output:



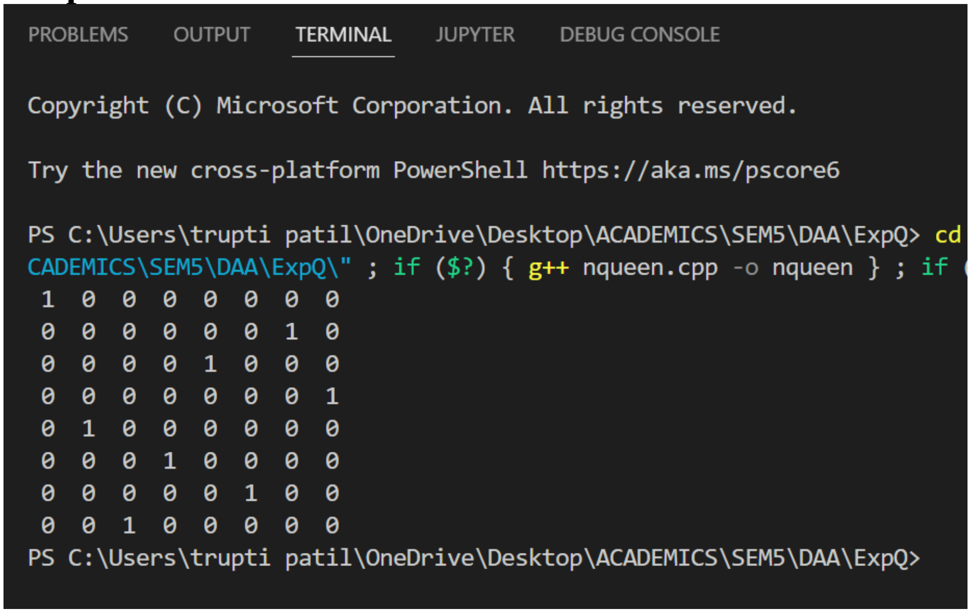
```
CADEMICS\SEM5\DAA\ExpQ\" ; if ($?) { g++ A10Q1a.cpp -o A10Q1a } ; if (
4
**Q*
Q***
***Q
*Q**

*Q**
***Q
Q***
**Q*

PS C:\Users\trupti patil\OneDrive\Desktop\ACADEMICS\SEM5\DAA\ExpQ>
```

Complexity Analysis:
Time Complexity: O(n^2)
Space complexity: O(n)

b) 8-Queen's problem
Algorithm:
This is optimized solution than the above solution.
Here we do not run loop to check if the place is safe or not.
Iterate over n rows and check using the formula if the place is safe or not.
Code:

```
#include<bits/stdc++.h>
using namespace std;
#define N 8

bool solveNQUtil(vector<vector<int>>& board, int col,vector<int>&
leftRow,
vector<int>& LeftDiagonal,vector<int>& RightDiagonal){
    if (col >= N)
        return true;

    for (int i = 0; i < N; i++) {
        if ((LeftDiagonal[i - col + N - 1] != 1 && RightDiagonal[i +
col] != 1) && leftRow[i] != 1) {
            board[i][col] = 1;
            LeftDiagonal[i - col + N - 1] = RightDiagonal[i + col] =
leftRow[i] = 1;

            if (solveNQUtil(board, col +
1,leftRow,LeftDiagonal,RightDiagonal))
                return true;

            board[i][col] = 0;
            LeftDiagonal[i - col + N - 1] = RightDiagonal[i + col] =
leftRow[i] = 0;
        }
    }
    return false;
}

int main(){
    int q=8;
    vector<vector<int>> board={{ 0, 0, 0, 0, 0, 0, 0, 0 },
                        { 0, 0, 0, 0, 0, 0, 0, 0 },
                        { 0, 0, 0, 0, 0, 0, 0, 0 },
                        { 0, 0, 0, 0, 0, 0, 0, 0 },
                        { 0, 0, 0, 0, 0, 0, 0, 0 },
                        { 0, 0, 0, 0, 0, 0, 0, 0 },
                        { 0, 0, 0, 0, 0, 0, 0, 0 },
                        { 0, 0, 0, 0, 0, 0, 0, 0 }};

    vector<int> leftrow(q,0);
    vector<int> LeftDiagonal(2*q-1,0);
    vector<int> RightDiagonal(2*q-1,0);

    if (solveNQUtil(board, 0,leftrow,LeftDiagonal,RightDiagonal) ==
false) {
        cout<<"Solution does not exist";
        return false;
    }

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
            cout<<" "<< board[i][j]<<" ";
        cout<<endl;
    }
    return 0;
}
```

Output:



PROBLEMS OUTPUT TERMINAL JUPYTER DEBUG CONSOLE

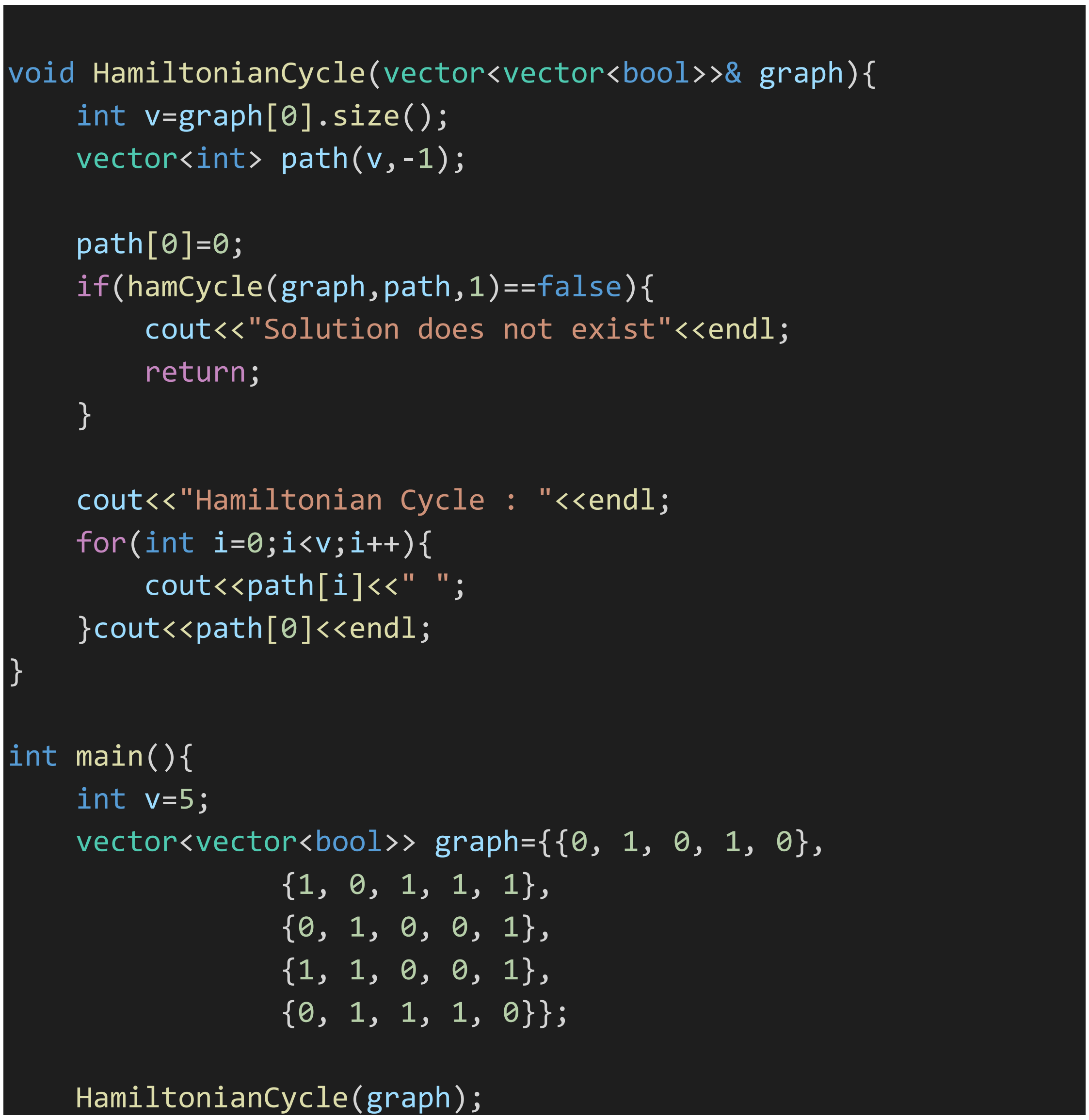
```
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\trupti patil\OneDrive\Desktop\ACADEMICS\SEM5\DAA\ExpQ> cd
CADEMICS\SEM5\DAA\ExpQ\" ; if ($?) { g++ nqueen.cpp -o nqueen } ; if
 1  0  0  0  0  0  0  0
 0  0  0  0  0  0  1  0
 0  0  0  0  1  0  0  0
 0  0  0  0  0  0  0  1
 0  1  0  0  0  0  0  0
 0  0  0  1  0  0  0  0
 0  0  0  0  0  1  0  0
 0  0  1  0  0  0  0  0
PS C:\Users\trupti patil\OneDrive\Desktop\ACADEMICS\SEM5\DAA\ExpQ>
```

Complexity Analysis:
Time Complexity: O(n!)
Space Complexity: O(n^2)

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

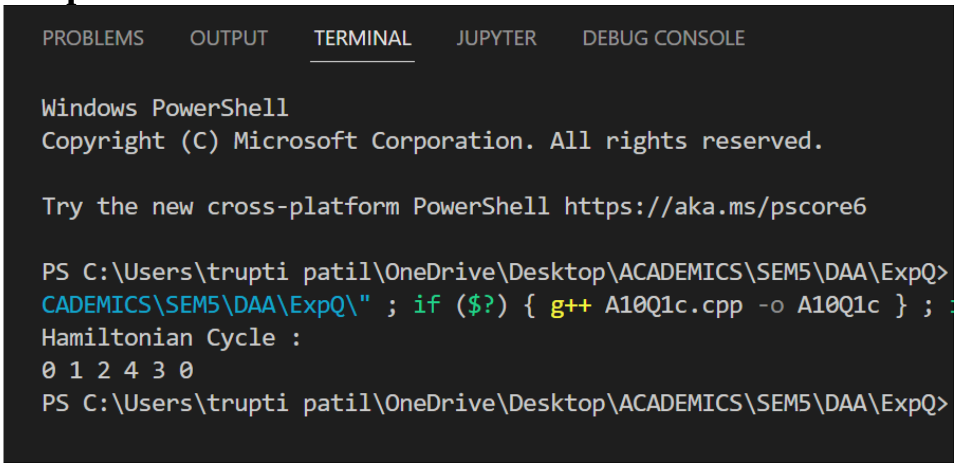
Code:

```
     // Hamiltanion Cycle

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

bool isSafe(int x,vector<vector<bool>>& graph,vector<int>&
path,int pos){
    // check if vertex is adjacent of previously added vertex
    if(graph[path[pos-1]][x]==0){
        return false;
    }

    // check if vertex is already added
    for(int i=0;i<pos;i++){
        if(path[i]==x){
            return false;
        }
    }
    return true;
}

bool hamCycle(vector<vector<bool>>& graph,vector<int>&
path,int pos){
    int v=graph[0].size();
    // base case
    if(pos==v){
        // check if there is edge between last vertex and
first vertex
        if(graph[path[pos-1]][path[0]]==1){
            return true;
        }else{
            return false;
        }
    }

    for(int i=1;i<v;i++){
        if(isSafe(i,graph,path,pos)){
            path[pos]=i;

            // recursive call
            if(hamCycle(graph,path,pos+1)==true){
                return true;
            }

            // if adding vertex i does not given any solution
then remove it
            path[pos]=-1;
        }
    }
    // if no vertex can be added to cycle constructed till now
    return false;
}
```



```cpp
void HamiltonianCycle(vector<vector<bool>>& graph){
    int v=graph[0].size();
    vector<int> path(v,-1);

    path[0]=0;
    if(hamCycle(graph,path,1)==false){
        cout<<"Solution does not exist"<<endl;
        return;
    }

    cout<<"Hamiltonian Cycle : "<<endl;
    for(int i=0;i<v;i++){
        cout<<path[i]<<" ";
    }cout<<path[0]<<endl;
}

int main(){
    int v=5;
    vector<vector<bool>> graph={{0, 1, 0, 1, 0},
                                {1, 0, 1, 1, 1},
                                {0, 1, 0, 0, 1},
                                {1, 1, 0, 0, 1},
                                {0, 1, 1, 1, 0}};

    HamiltonianCycle(graph);
```

Output:



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\trupti patil\OneDrive\Desktop\ACADEMICS\SEM5\DAA\ExpQ>
CADEMICS\SEM5\DAA\ExpQ\" ; if ($?) { g++ A10Q1c.cpp -o A10Q1c } ;
Hamiltonian Cycle :
0 1 2 4 3 0
PS C:\Users\trupti patil\OneDrive\Desktop\ACADEMICS\SEM5\DAA\ExpQ>
```

Complexity Analysis:

Time Complexity: O(n^n)

d) Graph colouring Problem

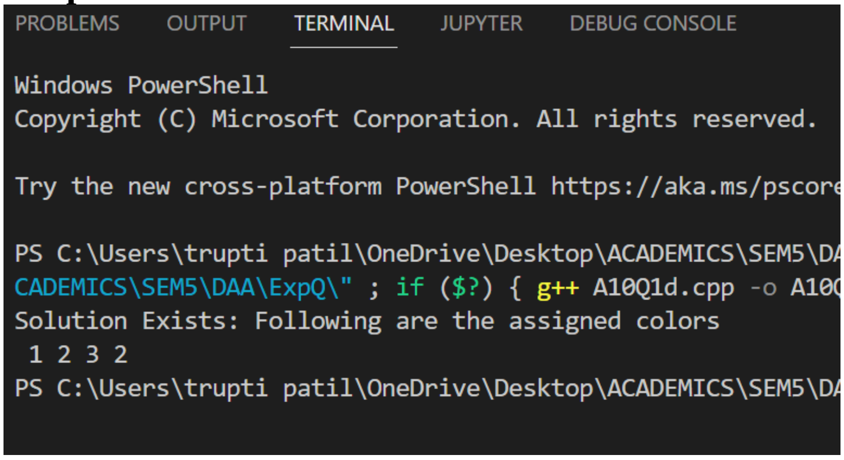
Algorithm:

1. Create a recursive function that takes the graph, current index, number of vertices, and output color array.
2. If the current index is equal to the number of vertices. Print the color configuration in the output array.
3. Assign a color to a vertex (1 to m).
4. For every assigned color, check if the configuration is safe, (i.e. check if the adjacent vertices do not have the same color) recursively call the function with the next index and number of vertices
5. If any recursive function returns true break the loop and return true
6. If no recursive function returns true then return false

Code:

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

#define V 4

void printSolution(int color[]);

bool isSafe(bool graph[V][V], int color[]){
    for (int i = 0; i < V; i++)
        for (int j = i + 1; j < V; j++)
            if (graph[i][j] && color[j] == color[i])
                return false;
    return true;
}

bool graphColoring(bool graph[V][V], int m, int i,
                int color[V])
{
    if (i == V) {

        if (isSafe(graph, color)) {

            printSolution(color);
            return true;
        }
        return false;
    }
    for (int j = 1; j <= m; j++) {
        color[i] = j;

        if (graphColoring(graph, m, i + 1, color))
            return true;

        color[i] = 0;
    }

    return false;
}

void printSolution(int color[])
{
    cout << "Solution Exists:"
            " Following are the assigned colors \n";
    for (int i = 0; i < V; i++)
        cout << " " << color[i];
    cout << "\n";
}

int main()
{
    bool graph[V][V] = {
        { 0, 1, 1, 1 },
        { 1, 0, 1, 0 },
        { 1, 1, 0, 1 },
        { 1, 0, 1, 0 },
    };
    int m = 3;
    int color[V];
    for (int i = 0; i < V; i++)
        color[i] = 0;

    if (!graphColoring(graph, m, 0, color))
        cout << "Solution does not exist";

    return 0;
}
```

Output:



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore

PS C:\Users\trupti patil\OneDrive\Desktop\ACADEMICS\SEM5\DA
CADEMICS\SEM5\DAA\ExpQ\" ; if ($?) { g++ A10Q1d.cpp -o A10
Solution Exists: Following are the assigned colors
 1 2 3 2
PS C:\Users\trupti patil\OneDrive\Desktop\ACADEMICS\SEM5\DA
```

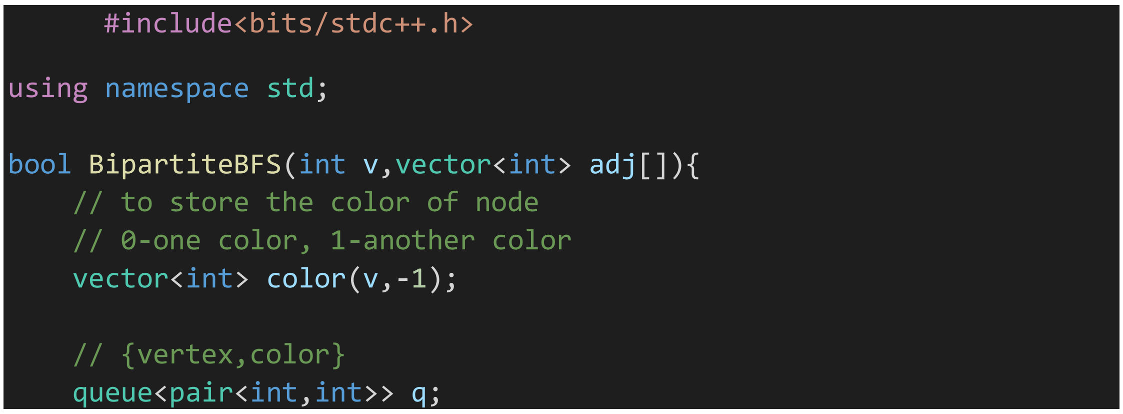
Complexity Analysis:
Time Complexity: O(m^V). There is a total O(m^V) combination of colors
Auxiliary Space: O(V). Recursive Stack of graph coloring (…) function will require O(V) space.

2) Implement following problem using graph traversal Technique

a) Check whether a graph is Bipartite or not using Breadth First Search (BFS)

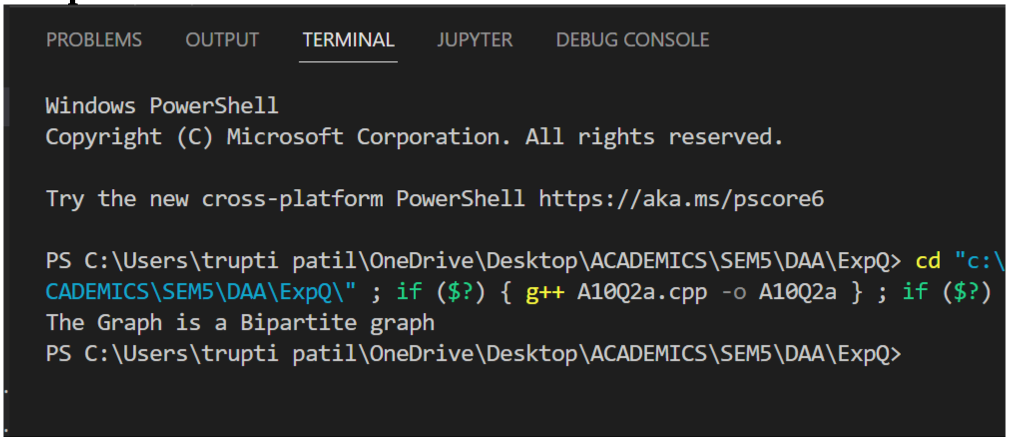
Algorithm:
1. Assign one color to the source vertex (putting into set one say U).
2. Color all the neighbors with another color (putting into other set V).
3. Color all neighbor’s neighbor with 1 color (putting into set U).
4. This way, assign color to all vertices such that it satisfies all the constraints of m way coloring problem where m = 2.
5. While assigning colors, if we find a neighbor which is colored with same color as current vertex, then the graph cannot be colored with 2 vertices (or graph is not Bipartite)

Code:



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

using namespace std;

bool BipartiteBFS(int v,vector<int> adj[]){
    // to store the color of node
    // 0-one color, 1-another color
    vector<int> color(v,-1);

    // {vertex,color}
    queue<pair<int,int>> q;
    // check for every vertex which is not visited
    // works for both connected and disconnected graph
    for(int i=0;i<v;i++){

        // vertex not colored
        if(color[i]==-1){
            q.push({i,0});
            color[i]=0;


            while(!q.empty()){
                pair<int,int> p=q.front();
                q.pop();

                // vertex
                int n=p.first;
                // colore of vertex
                int c=p.second;

                // iterate for adjacent vertices of current vertex
                for(int it: adj[n]){
                    // if color of adjacent node is same as of
current node
                    if(color[it]==c){
                        return false;
                    }// if adjacent node is not colored
                    else if(color[it]==-1){
                    // color adjacent node with color opposite to
current node
                        color[it]=(c) ? 0 : 1;
                        q.push({it,color[it]});
                    }
                }
            }
        }
    }
    return true;
}

int main(){
    int v=4,e=8;
    vector<int> adj[v]={{1,3},{0,2},{1,3},{0,2}};
    // vector<int> adj[v]={{0,1},{1,2},{2,3},{3,4},{4,0}};

    bool ans=BipartiteBFS(v,adj);

    if(ans){
        cout<<"The Graph is a Bipartite graph"<<endl;
    }else{
        cout<<"The Graph is not a Bipartite graph"<<endl;
    }

}
```

Output:



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\trupti patil\OneDrive\Desktop\ACADEMICS\SEM5\DAA\ExpQ> cd "c:\
CADEMICS\SEM5\DAA\ExpQ\" ; if ($?) { g++ A10Q2a.cpp -o A10Q2a } ; if ($?)
The Graph is a Bipartite graph
PS C:\Users\trupti patil\OneDrive\Desktop\ACADEMICS\SEM5\DAA\ExpQ>
```

Complexity Analysis:
Time Complexity: O(V+E)
Auxiliary Space: O(V)

b) Find Articulation Point in Graph using Depth First Search (DFS) and mention whether Graph is Biconnected or not.

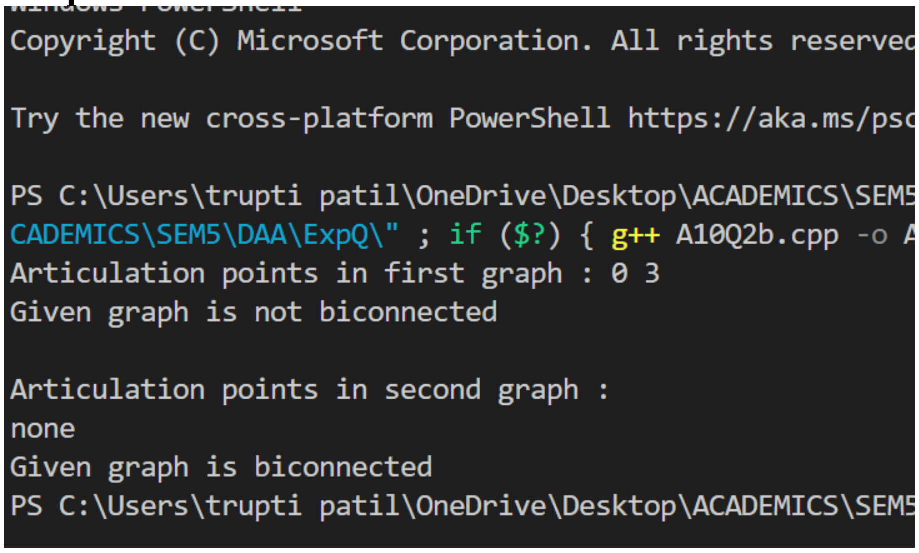
Algorithm:

1. Do DFS traversal of the given graph
2. In DFS traversal, maintain a parent[] array where parent[u] stores the parent of vertex u.
3. To check if u is the root of the DFS tree and it has at least two children. For every vertex, count children. If the currently visited vertex u is root (parent[u] is NULL) and has more than two children, print it.
4. To handle a second case where u is not the root of the DFS tree and it has a child v such that no vertex in the subtree rooted with v has a back edge to one of the ancestors in DFS tree of u maintain an array disc[] to store the discovery time of vertices.
5. For every node u, find out the earliest visited vertex (the vertex with minimum discovery time) that can be reached from the subtree rooted with u. So we maintain an additional array low[] such that: low[u] = min(disc[u], disc[w]) , Here w is an ancestor of u and there is a back edge from some descendant of u to w.

Code:

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

using namespace std;

void APUtil(vector<int> adj[], int u, bool visited[],int
disc[], int low[], int& time, int parent,bool isAP[]){
    int children = 0;

    visited[u] = true;
    disc[u] = low[u] = ++time;

    for (auto v : adj[u]) {
        if (!visited[v]) {
            children++;
            APUtil(adj, v, visited, disc, low, time, u, isAP);
            low[u] = min(low[u], low[v]);
            if (parent != -1 && low[v] >= disc[u])
                isAP[u] = true;
        }
        else if (v != parent)
            low[u] = min(low[u], disc[v]);
    }
    if (parent == -1 && children > 1)
        isAP[u] = true;
}

void AP(vector<int> adj[], int V){
    int disc[V] = { 0 };
    int low[V];
    bool visited[V] = { false };
    bool isAP[V] = { false };
    int time = 0, par = -1;

    for (int u = 0; u < V; u++){
        if (!visited[u]){
            APUtil(adj, u, visited, disc, low,time, par,
isAP);
        }
    }

    bool flag=false;
    for (int u = 0; u < V; u++){
        if (isAP[u] == true){
            flag=true;
            cout << u << " ";
        }
    }cout<<endl;

    if(flag){
        cout<<"Given graph is not biconnected"<<endl;
    }else{
        cout<<"none"<<endl;
        cout<<"Given graph is biconnected"<<endl;
    }
}


void addEdge(vector<int> adj[], int u, int v){
    adj[u].push_back(v);
    adj[v].push_back(u);
}

int main(){
    cout << "Articulation points in first graph : ";
    int V = 5;
    vector<int> adj1[V];
    addEdge(adj1, 1, 0);
    addEdge(adj1, 0, 2);
    addEdge(adj1, 2, 1);
    addEdge(adj1, 0, 3);
    addEdge(adj1, 3, 4);
    AP(adj1, V);

    cout << "\nArticulation points in second graph : ";
    V = 4;
    vector<int> adj2[V];
    addEdge(adj2, 0, 1);
    addEdge(adj2, 1, 2);
    addEdge(adj2, 2, 0);
    AP(adj2, V);

    return 0;
}
```

Output:



```
Copyright (C) Microsoft Corporation. All rights reservec

Try the new cross-platform PowerShell https://aka.ms/psc

PS C:\Users\trupti patil\OneDrive\Desktop\ACADEMICS\SEM5
CADEMICS\SEM5\DAA\ExpQ\" ; if ($?) { g++ A10Q2b.cpp -o A
Articulation points in first graph : 0 3
Given graph is not biconnected

Articulation points in second graph :
none
Given graph is biconnected
PS C:\Users\trupti patil\OneDrive\Desktop\ACADEMICS\SEM5
```

Complexity Analysis:
Time Complexity: O(V+E), For DFS it takes O(V+E) time.
Auxiliary Space: O(V+E), For visited array, adjacency list array.