ARTIFICIAL INTELLIGENCE

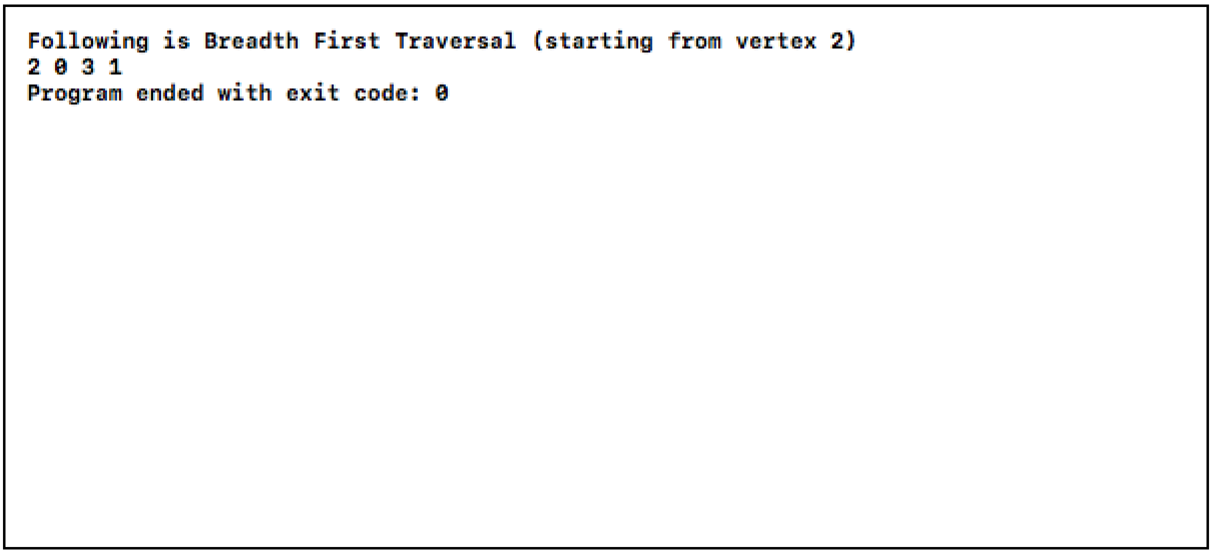
Submitted to:
Mr. Kanwalpreet Singh Malhi

Submitted by:
Sanu Kumar
SG - 15337
B.E. CSE 5th Sem

COMPUTER SCIENCE AND ENGINEERING

INDEX

Output: Breadth First Search



```
Following is Breadth First Traversal (starting from vertex 2)
2 0 3 1
Program ended with exit code: 0
```

Practical 1: Breadth First Search

```
#include<iostream>
#include <list>

using namespace std;

class Graph
{
   int V;
   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 = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
   adj[v].push_back(w);
}

void Graph::BFS(int s)
{
   bool *visited = new bool[V];
   for(int i = 0; i < V; i++)
      visited[i] = false;

   list<int> queue;

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

   list<int>::iterator i;

   while(!queue.empty())
   {
      s = queue.front();
```

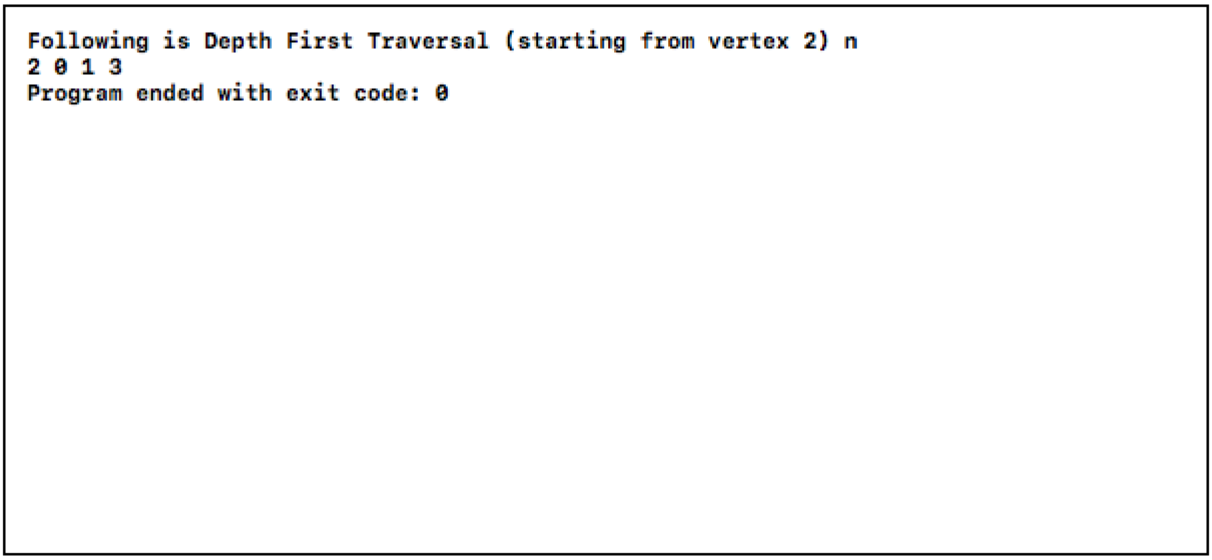
```
  cout << s << " ";
        queue.pop_front();

        for(i = adj[s].begin(); i != adj[s].end(); ++i)
        {
            if(!visited[*i])
            {
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }
}

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)" << endl;
    g.BFS(2);
    cout << endl;
    return 0;
}
```

OUTPUT: DEPTH FIRST SEARCH



```
Following is Depth First Traversal (starting from vertex 2) n
2 0 1 3
Program ended with exit code: 0
```

Practical 2: Depth First Search

```
#include<iostream>
#include<list>

using namespace std;

class Graph
{
    int V;
    list<int> *adj;
    void DFSUtil(int v, bool visited[]);
public:
    Graph(int V);
    void addEdge(int v, int w);
    void DFS(int v);
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);\
}

void Graph::DFSUtil(int v, bool visited[])
{\
    visited[v] = true;
    cout << v << " ";

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

void Graph::DFS(int v)
{
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;
```

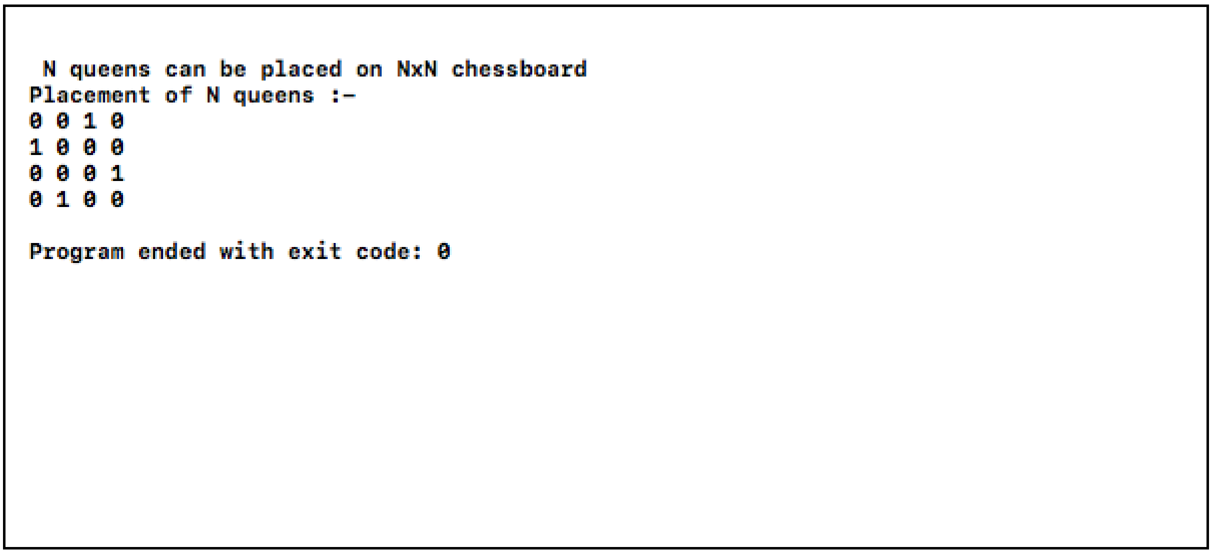
```
    DFSUtil(v, visited);
}

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 Depth First Traversal (starting from vertex 2) n" << endl;
    g.DFS(2);
    cout << endl;

    return 0;
}
```

Output: 8-Queen Problem



```
 N queens can be placed on NxN chessboard
Placement of N queens :-
0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0

Program ended with exit code: 0
```

Practical 3: 8-Queen Problem

```
#include<iostream>
#define N 4
using namespace std;

void printPlacement(int chess_board[N][N]) {
   int i,j;
   cout<<"\nPlacement of N queens :-\n";
   for (i = 0; i < N; i++) {
      for (j = 0; j < N; j++) {
         cout<<chess_board[i][j]<<" ";
      }
      cout<<endl;
   }
}

bool isCellSafe(int chess_board[N][N], int r_idx, int c_idx) {
   int i, j;

   for (i = 0; i < c_idx; i++) {
      if (chess_board[r_idx][i] == 1) {
         return false;
      }
   }

   i = r_idx; j = c_idx;
   while (i >= 0 && j >= 0) {
      if (chess_board[i][j] == 1) {
         return false;
      }
      i--; j--;
   }

   i = r_idx; j = c_idx;
   while (i < N && j >= 0) {
      if (chess_board[i][j] == 1) {
         return false;
      }
      i++; j--;
   }

   return true;
```

```
}

bool placeNQueens(int chess_board[N][N], int c_idx) {
   if (c_idx >= N) {
      return true;
   }

   int i;
   for (i = 0; i < N; i++) {
      if (isCellSafe(chess_board, i, c_idx)) {
         chess_board[i][c_idx] = 1;

         if (placeNQueens(chess_board, c_idx + 1) == true )
            return true;

         chess_board[i][c_idx] = 0;
      }
   }

   return false;
}


int main() {
   int chess_board[N][N] = { {0, 0, 0, 0},
      {0, 0, 0, 0},
      {0, 0, 0, 0},
      {0, 0, 0, 0} };
   bool n_queens_sol = placeNQueens(chess_board,0);
   if (n_queens_sol == false) {
      cout<<"\n N queens placement not possible";
   }
   else {
      cout<<"\n N queens can be placed on NxN chessboard";
      printPlacement(chess_board);
   }
   cout<<endl;
   return 0;
}
```