LAB ASSIGNMENT

1. Implement DFS algorithm

A. CODE:

```
#include <stdio.h>
#include <stdlib.h>

struct node {
  int vertex;
  struct node* next;
};

struct node* createNode(int v);

struct Graph
{
  int numVertices;
  int* visited;

  struct node** adjLists;
};


void DFS(struct Graph* graph, int vertex)
{
  struct node* adjList = graph->adjLists[vertex];
  struct node* temp = adjList;

  graph->visited[vertex] = 1;
  printf("Visited %d \n", vertex);

  while (temp != NULL)
  {
    int connectedVertex = temp->vertex;

    if (graph->visited[connectedVertex] == 0)
```

```
    {
      DFS(graph, connectedVertex);
    }
    temp = temp->next;
  }
}


struct node* createNode(int v)
{
  struct node* newNode = malloc(sizeof(struct node));
  newNode->vertex = v;
  newNode->next = NULL;
  return newNode;
}


struct Graph* createGraph(int vertices)
{
  struct Graph* graph = malloc(sizeof(struct Graph));
  graph->numVertices = vertices;

  graph->adjLists = malloc(vertices * sizeof(struct node*));

  graph->visited = malloc(vertices * sizeof(int));

  int i;
  for (i = 0; i < vertices; i++)
  {
    graph->adjLists[i] = NULL;
    graph->visited[i] = 0;
  }
  return graph;
}


void addEdge(struct Graph* graph, int src, int dest)
{
  struct node* newNode = createNode(dest);
  newNode->next = graph->adjLists[src];
```

```
  graph->adjLists[src] = newNode;


  newNode = createNode(src);
  newNode->next = graph->adjLists[dest];
  graph->adjLists[dest] = newNode;
}


void printGraph(struct Graph* graph)
{
  int v;
  for (v = 0; v < graph->numVertices; v++)
  {
    struct node* temp = graph->adjLists[v];
    printf("\n Adjacency list of vertex %d\n ", v);
    while (temp)
    {
      printf("%d -> ", temp->vertex);
      temp = temp->next;
    }
    printf("\n");
  }
}

int main()
{
  struct Graph* graph = createGraph(4);
  addEdge(graph, 0, 1);
  addEdge(graph, 0, 2);
  addEdge(graph, 1, 2);
  addEdge(graph, 2, 3);

  printGraph(graph);

  DFS(graph, 2);

  return 0;
}
```

SAMPLE INPUT AND SAMPLE OUTPUT:

```
Adjacency list of vertex 0
2 -> 1 ->

Adjacency list of vertex 1
2 -> 0 ->

Adjacency list of vertex 2
3 -> 1 -> 0 ->

Adjacency list of vertex 3
2 ->
Visited 2
Visited 3
Visited 1
Visited 0
```

Time Complexity Analysis:

Time Complexity of DFS is O(V + E), where V is the number of vertices and E is the number of edges in the graph when Adjacency List is used and O(V^2) when Adjacency Matrix is used.

2. Implement BFS algorithm

A. CODE:

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 40

struct queue
{
  int items[SIZE];
  int front;
  int rear;
};

struct queue* createQueue();
void enqueue(struct queue* q, int);
int dequeue(struct queue* q);
void display(struct queue* q);
int isEmpty(struct queue* q);
void printQueue(struct queue* q);

struct node
{
  int vertex;
  struct node* next;
};

struct node* createNode(int);

struct Graph
{
  int numVertices;
  struct node** adjLists;
  int* visited;
};


void bfs(struct Graph* graph, int startVertex)
{
```

```
  struct queue* q = createQueue();

  graph->visited[startVertex] = 1;
  enqueue(q, startVertex);

  while (!isEmpty(q))
  {
    printQueue(q);
    int currentVertex = dequeue(q);
    printf("Visited %d\n", currentVertex);

    struct node* temp = graph->adjLists[currentVertex];

    while (temp)
    {
      int adjVertex = temp->vertex;

      if (graph->visited[adjVertex] == 0)
      {
        graph->visited[adjVertex] = 1;
        enqueue(q, adjVertex);
      }
      temp = temp->next;
    }
  }
}


struct node* createNode(int v)
{
  struct node* newNode = malloc(sizeof(struct node));
  newNode->vertex = v;
  newNode->next = NULL;
  return newNode;
}


struct Graph* createGraph(int vertices)
{
  struct Graph* graph = malloc(sizeof(struct Graph));
```

```
  graph->numVertices = vertices;

  graph->adjLists = malloc(vertices * sizeof(struct node*));
  graph->visited = malloc(vertices * sizeof(int));

  int i;
  for (i = 0; i < vertices; i++)
  {
    graph->adjLists[i] = NULL;
    graph->visited[i] = 0;
  }

  return graph;
}


void addEdge(struct Graph* graph, int src, int dest)
{

  struct node* newNode = createNode(dest);
  newNode->next = graph->adjLists[src];
  graph->adjLists[src] = newNode;


  newNode = createNode(src);
  newNode->next = graph->adjLists[dest];
  graph->adjLists[dest] = newNode;
}


struct queue* createQueue()
{
  struct queue* q = malloc(sizeof(struct queue));
  q->front = -1;
  q->rear = -1;
  return q;
}


int isEmpty(struct queue* q)
```

```
{
  if (q->rear == -1)
    return 1;
  else
    return 0;
}


void enqueue(struct queue* q, int value)
{
  if (q->rear == SIZE - 1)
    printf("\nQueue is Full!!");
  else
  {
    if (q->front == -1)
      q->front = 0;
    q->rear++;
    q->items[q->rear] = value;
  }
}


int dequeue(struct queue* q)
{
  int item;
  if (isEmpty(q))
  {
    printf("Queue is empty");
    item = -1;
  }
  else
  {
    item = q->items[q->front];
    q->front++;
    if (q->front > q->rear)
    {
      printf("Resetting queue ");
      q->front = q->rear = -1;
    }
  }
```

```
  return item;
}


void printQueue(struct queue* q)
{
  int i = q->front;

  if (isEmpty(q))
  {
    printf("Queue is empty");
  }
  else
  {
    printf("\nQueue contains \n");
    for (i = q->front; i < q->rear + 1; i++)
    {
      printf("%d ", q->items[i]);
    }
  }
}

int main()
{
  struct Graph* graph = createGraph(6);
  addEdge(graph, 0, 1);
  addEdge(graph, 0, 2);
  addEdge(graph, 1, 2);
  addEdge(graph, 1, 4);
  addEdge(graph, 1, 3);
  addEdge(graph, 2, 4);
  addEdge(graph, 3, 4);

  bfs(graph, 0);

  return 0;
}
```

SAMPLE INPUT AND SAMPLE OUTPUT:

```
Queue contains
0 Resetting queue Visited 0

Queue contains
2 1 Visited 2

Queue contains
1 4 Visited 1

Queue contains
4 3 Visited 4

Queue contains
3 Resetting queue Visited 3
```

Time Complexity Analysis:

The Time complexity of BFS is O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges.

3. Implement the N-Queen problem.

A. CODE:

```
#include<stdio.h>
#include<math.h>

int board[20],count;

int main()
{
int n,i,j;
void queen(int row,int n);

printf(" - N Queens Problem Using Backtracking -");
printf("\n\nEnter number of Queens:");
scanf("%d",&n);
queen(1,n);
return 0;
}


void print(int n)
{
int i,j;
printf("\n\nSolution %d:\n\n",++count);

for(i=1;i<=n;++i)
  printf("\t%d",i);

for(i=1;i<=n;++i)
{
  printf("\n\n%d",i);
  for(j=1;j<=n;++j)
  {
   if(board[i]==j)
    printf("\tQ");
   else
    printf("\t-");
  }
```

```
}
}


int place(int row,int column)
{
int i;
for(i=1;i<=row-1;++i)
{

  if(board[i]==column)
   return 0;
  else
   if(abs(board[i]-column)==abs(i-row))
    return 0;
}

return 1;
}


void queen(int row,int n)
{
int column;
for(column=1;column<=n;++column)
{
  if(place(row,column))
  {
   board[row]=column;
   if(row==n)
    print(n);
   else
    queen(row+1,n);
  }
}
}
```

SAMPLE INPUT AND SAMPLE OUTPUT:

```
 - N Queens Problem Using Backtracking -

Enter number of Queens:4


Solution 1:

        1       2       3       4

1       -       Q       -       -

2       -       -       -       Q

3       Q       -       -       -

4       -       -       Q       -
```

```
Solution 2:

        1       2       3       4

1       -       -       Q       -

2       Q       -       -       -

3       -       -       -       Q

4       -       Q       -       -
```

Time Complexity Analysis:

The worst case “brute force” solution for the N-queens puzzle has an O(n^n) time complexity. This means it will look through every position on an NxN board, N times, for N queens. ... This is over 100 times as fast as brute force and has a time complexity of O(2^n).

4. Solve Knight Tour Problem.

A. CODE:

```
#include <stdio.h>
#define N 8

int is_valid(int i, int j, int sol[N+1][N+1])
{
  if (i>=1 && i<=N && j>=1 && j<=N && sol[i][j]==-1)
    return 1;
  return 0;
}

int knight_tour(int sol[N+1][N+1], int i, int j, int step_count, int x_move[], int y_move[])
{
  if (step_count == N*N)
    return 1;

  int k;
  for(k=0; k<8; k++)
  {
    int next_i = i+x_move[k];
    int next_j = j+y_move[k];

    if(is_valid(i+x_move[k], j+y_move[k], sol))
    {
      sol[next_i][next_j] = step_count;
      if (knight_tour(sol, next_i, next_j, step_count+1, x_move, y_move))
        return 1;
      sol[i+x_move[k]][j+y_move[k]] = -1; // backtracking
    }
  }

  return 0;
}

int start_knight_tour()
{
  int sol[N+1][N+1];
```

```
  int i, j;
  for(i=1; i<=N; i++)
  {
    for(j=1; j<=N; j++)
    {
      sol[i][j] = -1;
    }
  }

  int x_move[] = {2, 1, -1, -2, -2, -1, 1, 2};
  int y_move[] = {1, 2, 2, 1, -1, -2, -2, -1};

  sol[1][1] = 0; // placing knight at cell(1, 1)

  if (knight_tour(sol, 1, 1, 1, x_move, y_move))
  {
    for(i=1; i<=N; i++)
    {
      for(j=1; j<=N; j++)
      {
        printf("%d\t",sol[i][j]);
      }
      printf("\n");
    }
    return 1;
  }
  return 0;
}

int main()
{
  printf("%d\n",start_knight_tour());
  return 0;
}
```

SAMPLE INPUT AND SAMPLE OUTPUT:

```
0   59  38  33  30  17  8   63
37  34  31  60  9   62  29  16
58  1   36  39  32  27  18  7
35  48  41  26  61  10  15  28
42  57  2   49  40  23  6   19
47  50  45  54  25  20  11  14
56  43  52  3   22  13  24  5
51  46  55  44  53  4   21  12
1
```

Time Complexity Analysis:

There are N*N i.e., N^2 cells in the board and we have a maximum of 8 choices to make from a cell,so the worst case Time Complexity is $O(8^{N^2})$