In [3]:

```
#Shivam Srivastava - 2101234EC
#1
import numpy as np
import random
from time import sleep

# Creates an empty board


def create_board():
    return(np.array([[0, 0, 0],
                     [0, 0, 0],
                     [0, 0, 0]]))

# Check for empty places on board


def possibilities(board):
    l = []

    for i in range(len(board)):
        for j in range(len(board)):

            if board[i][j] == 0:
                l.append((i, j))
    return(l)

# Select a random place for the player


def random_place(board, player):
    selection = possibilities(board)
    current_loc = random.choice(selection)
    board[current_loc] = player
    return(board)

# Checks whether the player has three
```

```
# of their marks in a horizontal row

def row_win(board, player):
    for x in range(len(board)):
        win = True

        for y in range(len(board)):
            if board[x, y] != player:
                win = False
                continue

        if win == True:
            return(win)
    return(win)

# Checks whether the player has three
# of their marks in a vertical row


def col_win(board, player):
    for x in range(len(board)):
        win = True

        for y in range(len(board)):
            if board[y][x] != player:
                win = False
                continue

        if win == True:
            return(win)
    return(win)

# Checks whether the player has three
# of their marks in a diagonal row


def diag_win(board, player):
    win = True
    y = 0
```

```
    for x in range(len(board)):
        if board[x, x] != player:
            win = False
    if win:
        return win
    win = True
    if win:
        for x in range(len(board)):
            y = len(board) - 1 - x
            if board[x, y] != player:
                win = False
    return win

# Evaluates whether there is
# a winner or a tie


def evaluate(board):
    winner = 0

    for player in [1, 2]:
        if (row_win(board, player) or
                col_win(board, player) or
                diag_win(board, player)):

            winner = player

    if np.all(board != 0) and winner == 0:
        winner = -1
    return winner

# Main function to start the game


def game():
    board, winner, counter = create_board(), 0, 1
    print(board)
    sleep(2)
```

```
    while winner == 0:
        for player in [1, 2]:
            board = random_place(board, player)
            print("Board after " + str(counter) + " move")
            print(board)
            sleep(2)
            counter += 1
            winner = evaluate(board)
            if winner != 0:
                break
    return(winner)


# Driver Code
print("Winner is: " + str(game()))
```

```
[[0 0 0]
 [0 0 0]
 [0 0 0]]
Board after 1 move
[[0 0 0]
 [0 0 1]
 [0 0 0]]
Board after 2 move
[[0 0 0]
 [0 0 1]
 [2 0 0]]
Board after 3 move
[[0 0 0]
 [0 1 1]
 [2 0 0]]
Board after 4 move
[[2 0 0]
 [0 1 1]
 [2 0 0]]
Board after 5 move
[[2 1 0]
 [0 1 1]
 [2 0 0]]
Board after 6 move
[[2 1 0]
 [2 1 1]
 [2 0 0]]
Winner is: 2
```

In [4]:

```
#2
N = 4

# ld is an array where its indices indicate row-col+N-1
ld = [0] * 30

# rd is an array where its indices indicate row+col
rd = [0] * 30

# Column array where its indices indicate column
cl = [0] * 30

# A utility function to print solution


def printSolution(board):
        for i in range(N):
                for j in range(N):
                        print(" Q " if board[i][j] == 1 else " . ",
end="")
                print()

# A recursive utility function to solve N Queen problem


def solveNQUtil(board, col):
        # Base case: If all queens are placed, return true
        if col >= N:
                return True

        for i in range(N):
                if (ld[i - col + N - 1] != 1 and rd[i + col] != 1) and
cl[i] != 1:
                        board[i][col] = 1
                        ld[i - col + N - 1] = rd[i + col] = cl[i] = 1
                        if solveNQUtil(board, col + 1):
                                return True
```

```
                        # If placing the queen in board[i][col] doesn't lead to a solution, backtrack
                        board[i][col] = 0 # BACKTRACK
                        ld[i - col + N - 1] = rd[i + col] = cl[i] = 0


def solveNQ():
        board = [[0 for _ in range(N)] for _ in range(N)]

        if not solveNQUtil(board, 0):
                print("Solution does not exist")
                return False

        printSolution(board)
        return True

if __name__ == "__main__":
        solveNQ()
```

```
 .  .  Q  .
 Q  .  .  .
 .  .  .  Q
 .  Q  .  .
```

In [7]:

```
#3
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}

visited = [] # List for visited nodes.
queue = []     #Initialize a queue

def bfs(visited, graph, node): #function for BFS
  visited.append(node)
  queue.append(node)

  while queue:          # Creating loop to visit each node
    m = queue.pop(0)
    print (m, end = " ")

    for neighbour in graph[m]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)

print("Following is the Breadth-First Search")
bfs(visited, graph, '5')
```

```
Following is the Breadth-First Search
5 3 7 2 4 8
```

In [8]:

```
#4
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}

visited = set() # Set to keep track of visited nodes of graph.

def dfs(visited, graph, node):  #function for dfs
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)

# Driver Code
print("Following is the Depth-First Search")
dfs(visited, graph, '5')
```

```
Following is the Depth-First Search
5
3
2
4
8
7
```

In [ ]: