Assignment 01 (AND and XOR operation) Roll No.335
Batch: B
Code:
```
def process_string(string, operation): result = ""
   for char in string:
      if operation == 'AND':
         result += chr(ord(char) & 127)
      elif operation == 'XOR':
          result += chr(ord(char) ^ 127)
   return result
input_string = "Hello, World!" print("Input String:", input_string)
result = process_string(input_string, 'AND') print("Result (AND):", result)
result = process_string(input_string, 'XOR') print("Result (XOR):", result)
```

Assignment No. 02(Rail Fence Cipher) Roll No.335
Batch: B
Code:
```
import math
# Encryption function for Rail Fence Cipher
def encryptRailFence(message, key):
message = message.replace(" ", "") # remove spaces from message num_rows = key
 num_cols = math.ceil(len(message) / num_rows)
arr = [[' ' for j in range(num_cols)] for i in range(num_rows)] k=0
for j in range(num_cols):
      for i in range(num_rows):
         if k < len(message):
            arr[i][j] = message[k]
k += 1 else:
break
   ciphertext = ""
   for i in range(num_rows):
      for j in range(num_cols):
         ciphertext += arr[i][j]
   return ciphertext
# Decryption function for Rail Fence Cipher
def decryptRailFence(ciphertext, key):
ciphertext = ciphertext.replace(" ", "") # remove spaces from ciphertext num_rows = key
num_cols = math.ceil(len(ciphertext) / num_rows)
arr = [[' ' for j in range(num_cols)] for i in range(num_rows)]
k=0
for i in range(num_rows):
      for j in range(num_cols):
         if k < len(ciphertext):
            arr[i][j] = ciphertext[k]
k += 1 else:
break
message = ""
k=0
for j in range(num_cols):
   for i in range(num_rows):
```

```
        if k < len(ciphertext):
            if arr[i][j] != ' ':
                message += arr[i][j]
                k += 1
else: break
return message
# Main code to take input from user and test the functions
message = input("Enter the message to encrypt: ") key = int(input("Enter the key: "))
ciphertext = encryptRailFence(message, key) print("Encrypted message:", ciphertext)
decrypted = decryptRailFence(ciphertext, key) print("Decrypted message:", decrypted)
```

Assignment No. 04(RSA algorithm) Roll No.335
Batch: B
Code:

```
import math
message = int(input("Enter the message to be encrypted: "))
p=7
q = 17 n = p*q
 m = (p-1)*(q-1)
for i in range(2,m):
   if math.gcd(i,m) == 1:
e=i
       break
for i in range(m):
if (e*i) % m == 1: d=i
break
def encrypt(me):
   c = pow(message, e, n)
   return c
def decrypt(ct):
   p = pow(ct, d, n)
   return p
print("Original Message is: ", message)
CT = encrypt(message)
print("Encrypted Message is: ", CT)
PT = decrypt(CT)
print("Decrypted Message is:", PT)
```

Assignment No.5(Diffi-Hellman Algorithm) Roll No. 335
Batch: B
Code:
HTML File:

```
 <!DOCTYPE html>
<html>
<head>
 <title>Diffie-Hellman Key Exchange</title> </head>
<body>
<h1>Diffie-Hellman Key Exchange</h1>
<p>Enter a prime number and a base value:</p> <form>
```

```
<label for="prime">Prime number:</label>
<input type="number" id="prime" name="prime"><br><br>
<label for="base">Base value:</label>
<input type="number" id="base" name="base"><br><br>
<button type="button" onclick="generateKeys()">Generate Keys</button>
</form>
<p>Public keys:</p>
<p>Alice: <span id="alicePubKey"></span></p> <p>Bob: <span id="bobPubKey"></span></p>
<p>Shared secret:</p>
<p><span id="sharedSecret"></span></p>
<script src="script.js"></script> </body>
</html>
```

Jscript File:

```
 function isPrime(n) {
if (n < 2) return false;
for (let i = 2; i <= Math.sqrt(n); i++) {
       if (n % i === 0) return false;
   }
   return true;
}
function generateKeys() {
const prime = parseInt(document.getElementById("prime").value); const base =
parseInt(document.getElementById("base").value);
if (!isPrime(prime)) {
alert("Please enter a prime number."); return;
}
const alicePrivateKey = Math.floor(Math.random() * (prime - 2)) + 2;
const bobPrivateKey = Math.floor(Math.random() * (prime - 2)) + 2; const alicePublicKey =
modExp(base, alicePrivateKey, prime);
 const bobPublicKey = modExp(base, bobPrivateKey, prime);
const sharedSecret = modExp(alicePublicKey, bobPrivateKey, prime);
document.getElementById("alicePubKey").textContent = alicePublicKey;
document.getElementById("bobPubKey").textContent = bobPublicKey;
document.getElementById("sharedSecret").textContent = sharedSecret;
}
function modExp(base, exponent, modulus) {
if (modulus === 1) return 0;
   let result = 1;
   base = base % modulus;
   while (exponent > 0) {
if (exponent % 2 === 1) {
result = (result * base) % modulus;
}
exponent = Math.floor(exponent / 2); base = (base * base) % modulus;
}
   return result;
}
```

Assignment No. 01(BFS & DFS) Roll No. 335
Batch: B

Code:

```
graph = {
  '1' : ['2','5'],
  '2' : ['3', '4'],
  '5' : ['6'],
  '3' : [],
  '4' : ['6'],
  '6' : [] }
visited = []
queue = []
def breadthFirstSearch(visited, graph, node): visited.append(node)
queue.append(node)
  while queue:
    m = queue.pop(0)
    print (m, end = " ")
    for neighbour in graph[m]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)
print("Breadth-First Search: ") breadthFirstSearch(visited, graph, '1')
# Program Output:-
# Breadth-First Search: #125346
# Depth First Search:
graph = {
  '1' : ['2','5'],
  '2' : ['3', '4'],
  '5' : ['6'],
  '3' : [],
  '4' : ['6'],
  '6' : []
}
visited = set()
def depthFirstSearch(visited, graph, node): if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in graph[node]:
depthFirstSearch(visited, graph, neighbour)
print("Depth-First Search") depthFirstSearch(visited, graph, '1')
```

Assignment No.02 (Selection Sort) Roll No. 335
Batch : B
Code:

```
def selection_sort_greedy(arr):
    n = len(arr)
print("\nList before Sorting: ", arr,"\n") for i in range(n):
        min_idx = i
        for j in range(i+1, n):
            if arr[j] < arr[min_idx]:
              min_idx = j
```

```
arr[i], arr[min_idx] = arr[min_idx], arr[i]
        print("List After Pass ",i+1,": ",arr)
    return arr
n=int(input("Length of List: "))
arr=[]
for i in range(n):
element=int(input("Enter List Element: "))
arr.append(element)
print("\nSorted List is:", selection_sort_greedy(arr))
```

Assignment No. 03(A star Algorithm) Roll No. 335
Batch: B
Code:

```
class box ( ):
"""A box class for A* Pathfinding"""
def __init__(self, parent=None, position=None): self.parent = parent
self.position = position
          self.g = 0
        self.h = 0
        self.f = 0
    def __eq__(self, other):
        return self.position == other.position
def astar(maze, start, end):
"""Returns a list of tuples as a path from the given start to the given
end in the given board"""
    # Create start and end node
start_node = box (None, start)
start_node.g = start_node.h = start_node.f = 0 end_node = box (None, end)
end_node.g = end_node.h = end_node.f = 0
    # Initialize both open and closed list
    open_list = []
    closed_list = []
    # Add the start node
    open_list.append (start_node)
    # Loop until you find the end
while len (open_list) > 0: # Get the current node
current_node = open_list[0]
current_index = 0
for index, item in enumerate (open_list):
    if item.f < current_node.f:
        current_node = item
        current_index = index
# Pop current off open list, add to closed list
open_list.pop (current_index)
closed_list.append (current_node)
# Found the goal
if current_node == end_node:
path = []
            current = current_node
            while current is not None:
```

```
            path.append (current.position)
current = current.parent
return path[::-1] # Return reversed path
    # Generate children
children = []
for new_position in [(0, -1), (0, 1), (-1, 0), (1, 0), (-1, -1), (-1, 1), (1, -1), (1, 1)]: # Adjacent squares
      # Get node position
node_position = (current_node.position[0] + new_position[0], current_node.position[1] +
new_position[1])
       # Make sure within range
if node_position[0] > (len (maze) - 1) or node_position[0] < 0 or node_position[1] > (
len (maze[len (maze) - 1]) - 1) or node_position[1] < 0: continue
      # Make sure walkable terrain
if maze[node_position[0]][node_position[1]] != 0: continue
      # Create new node
new_node = box (current_node, node_position)
# Append
      children.append (new_node)
    # Loop through children
    for child in children:
      # Child is on the closed list
      for closed_child in closed_list:
        if child == closed_child:
continue
      # Create the f, g, and h values
child.g = current_node.g + 1
child.h = ((child.position[0] - end_node.position[0]) ** 2) + (
(child.position[1] - end_node.position[1]) ** 2)
child.f = child.g + child.h
# Child is already in the open list
for open_node in open_list:
if child == open_node and child.g > open_node.g:
continue
# Add the child to the open list
open_list.append (child)
def main():
board = [[0, 0, 0, 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, 0],
      [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
      [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
      [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
      [0, 0, 0, 0, 1, 0, 0, 0, 0, 0]]
  start = (0, 0)
  end = (6, 6)
  path = astar (board, start, end)
  print (path)
  if __name__ == '__main__':
main ( )
```

Assignment No.5 (chatbot) Roll No. 335
Batch: B
Code:

```
import random
# Define possible chatbot responses
responses = {
"hello": ["Hi there!", "Hello!", "Hi!"],
"how are you": ["I'm doing well, thank you!", "Great, thanks for
 asking!", "I'm fine, how are you?"],
"what's your name": ["My name is Chatbot!", "I'm Chatbot, nice to meet
you!"],
"bye": ["Goodbye!", "See you later!", "Bye!"],
"default": ["I'm sorry, I didn't understand that.", "Could you please
repeat that?"]
}
# Define function to generate chatbot response
def generate_response(message):
  message = message.lower ( )
  for key in responses:
    if key in message:
return random.choice (responses[key]) return random.choice (responses["default"])
# Define main function to handle user input and chatbot response
def main():
print ("Hello! I'm a chatbot. What's your name?")
name = input ( )
print (f"Nice to meet you, {name}! How can I assist you today?")
  while True:
    message = input ( )
    if message.lower ( ) == "bye":
      print (generate_response ("bye"))
      break
    print (generate_response (message))
if __name__ == '__main__':
  main ( )
```

Assignment: 04(CSP N- queen problem) Roll No: 335
Batch: B
Code:

```
""" Python3 program to solve N Queen Problem
using Branch or Bound """
N=8
""" A utility function to print solution """
 def printSolution(board):
  for i in range (N):
    for j in range (N):
      print (board[i][j], end=" ")
    print ( )
""" A Optimized function to check if
a queen can be placed on board[row][col] """
```

```
def isSafe(row, col, slashCode, backslashCode, rowLookup, slashCodeLookup,
backslashCodeLookup):
if (slashCodeLookup[slashCode[row][col]] or
backslashCodeLookup[backslashCode[row][col]] or
        rowLookup[row]):
      return False
   return True
""" A recursive utility function
to solve N Queen problem """
def solveNQueensUtil(board, col, slashCode, backslashCode, rowLookup, slashCodeLookup,
            backslashCodeLookup):
""" base case: If all queens are placed then return True """
if (col >= N):
   return True
for i in range (N):
if (isSafe (i, col, slashCode, backslashCode, rowLookup, slashCodeLookup,
            backslashCodeLookup)):
      """ Place this queen in board[i][col] """
      board[i][col] = 1
      rowLookup[i] = True
slashCodeLookup[slashCode[i][col]] = True backslashCodeLookup[backslashCode[i][col]] = True
""" recur to place rest of the queens """
if (solveNQueensUtil (board, col + 1,
                 slashCode, backslashCode,
                 rowLookup, slashCodeLookup,
                 backslashCodeLookup)):
return True
""" If placing queen in board[i][col]
doesn't lead to a solution,then backtrack """
""" Remove queen from board[i][col] """
board[i][col] = 0
 rowLookup[i] = False slashCodeLookup[slashCode[i][col]] = False
backslashCodeLookup[backslashCode[i][col]] = False
   """ If queen can not be place in any row in
   this column col then return False """
   return False
def solveNQueens():
   board = [[0 for i in range (N)]
         for j in range (N)]
   # helper matrices
   slashCode = [[0 for i in range (N)]
            for j in range (N)]
   backslashCode = [[0 for i in range (N)]
               for j in range (N)]
   # arrays to tell us which rows are occupied
   rowLookup = [False] * N
# keep two arrays to tell us
# which diagonals are occupied x=2*N-1
slashCodeLookup = [False] * x backslashCodeLookup = [False] * x
   # initialize helper matrices
   for rr in range (N):
```

```
    for cc in range (N):
        slashCode[rr][cc] = rr + cc
        backslashCode[rr][cc] = rr - cc + 7
if (solveNQueensUtil (board, 0, slashCode, backslashCode, rowLookup, slashCodeLookup,
                    backslashCodeLookup) == False):
    print ("Solution does not exist")
return False
# solution found
printSolution (board)
return True
# Driver Code
solveNQueens ( )
# This code is contributed by SHUBHAMSINGH10
```