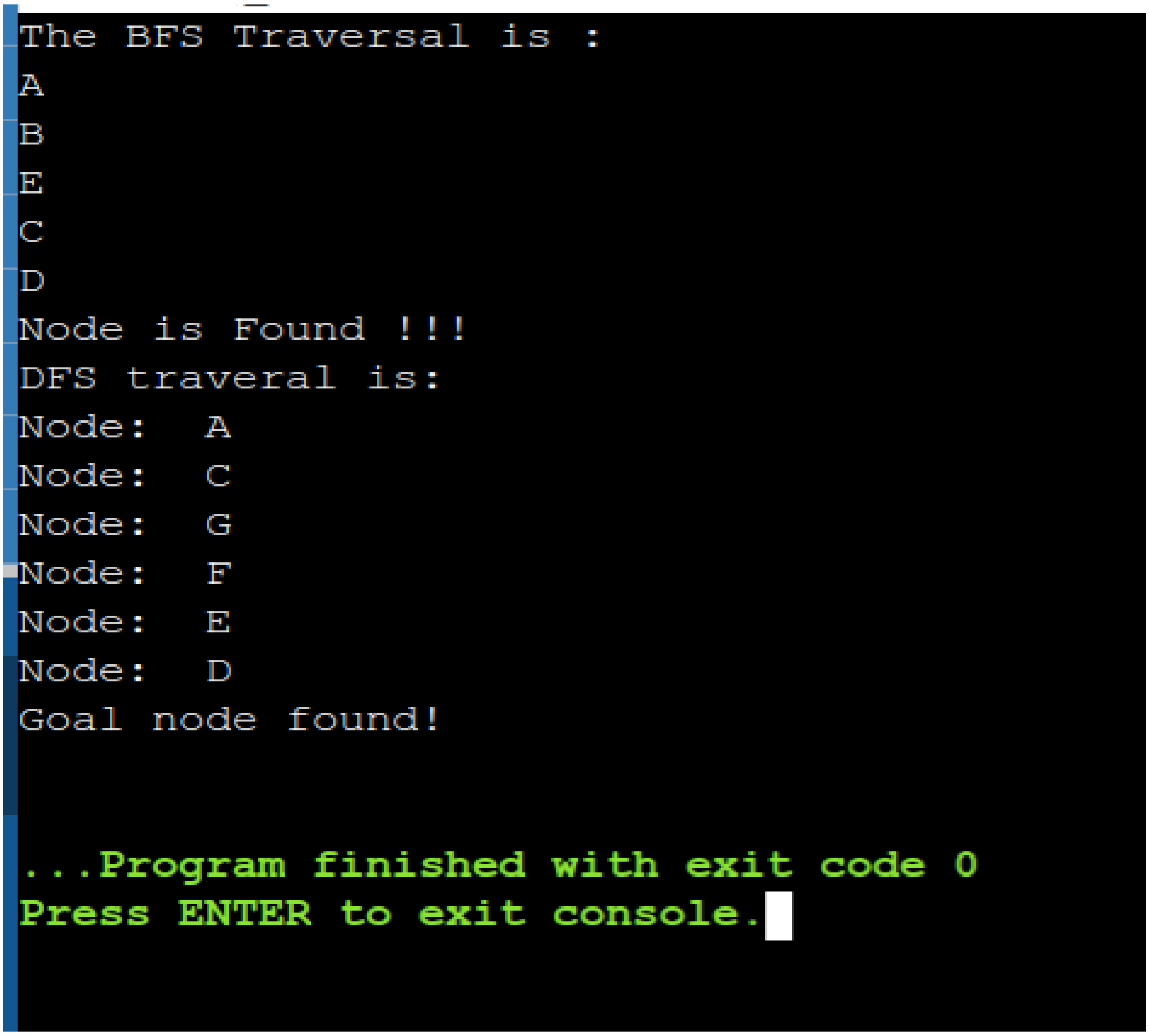
ASSIGNMENT 01:

```
# BFS
graph = {'A':['B', 'E', 'C'],
         'B':['A', 'D', 'E'],
         'D':['B', 'E'],
         'E':['A', 'D', 'B'],
         'C':['A', 'F', 'G'],
         'F':['C'],
         'G':['C']
         }
visited = []
queue = []
def bfs(visited, graph, start_node, goal_node):
    visited.append(start_node)
    queue.append(start_node)
    while queue:
        m = queue.pop(0)
        print(m)
        if m == goal_node:
            print("Node is Found !!! ")
            break
        else:
            for n in graph[m]:
                if n not in visited:
                    visited.append(n)
                    queue.append(n)
print("The BFS Traversal is : ")
bfs(visited, graph, 'A', 'D')

#DFS
graph = {'A':['B', 'E', 'C'],
         'B':['A', 'D', 'E'],
         'D':['B', 'E'],
         'E':['A', 'D', 'B'],
         'C':['A', 'F', 'G'],
         'F':['C'],
         'G':['C']
         }
visited = []
```

```
stack = []
def dfs(graph, start, goal):
    print("DFS traveral is: ")
    stack.append(start)
    visited.append(start)
    while stack:
        node = stack[-1]
        stack.pop()
        print("Node: ", node)
        if node == goal:
            print("Goal node found!")
            return
        for n in graph[node]:
            if n not in visited:
                visited.append(n)
                stack.append(n)

dfs(graph, 'A', "D")
```

OUTPUT:

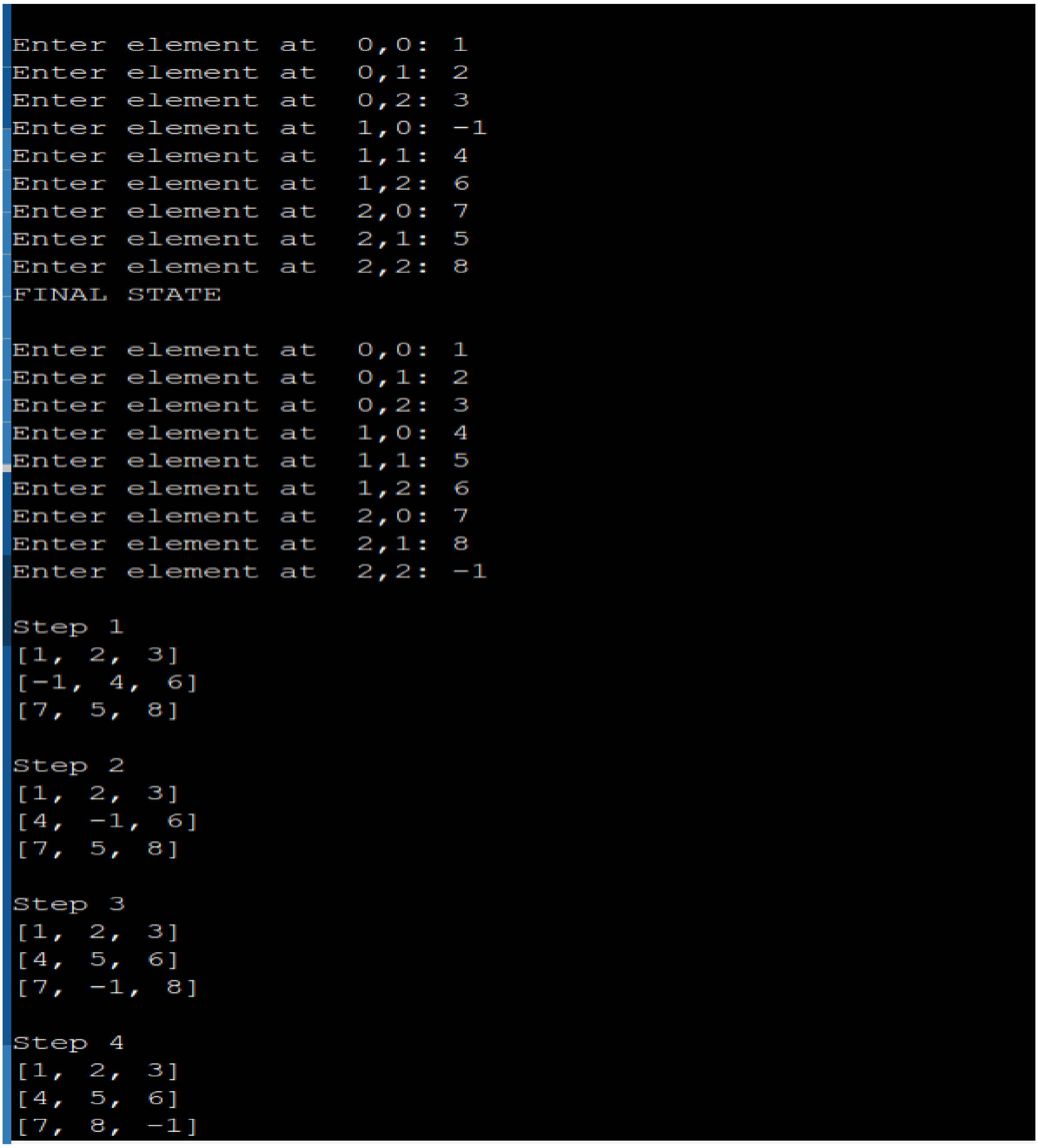


```
The BFS Traversal is :
A
B
E
C
D
Node is Found !!!
DFS traveral is:
Node:  A
Node:  C
Node:  G
Node:  F
Node:  E
Node:  D
Goal node found!


...Program finished with exit code 0
Press ENTER to exit console.
```

ASSIGNMENT 02:

```
import copy
final = [[1,2,3],[4,5,6],[7,8,-1]]
initial = [[1,2,3],[-1,4,6],[7,5,8]]
#function to find heuristic cost
def gn(state, finalstate):
    count = 0
for i in range(3):
        for j in range(3):
            if(state[i][j]!=-1):
                if(state[i][j] != finalstate[i][j]):
                    count+=1
    return count
def findposofblank(state):
    for i in range(3):
        for j in range(3):
            if(state[i][j] == -1):
                return [i,j]
def move_left(state, pos):
    if(pos[1]==0):
        return None
    retarr = copy.deepcopy(state)
    retarr[pos[0]][pos[1]],retarr[pos[0]][pos[1]-1] =
retarr[pos[0]][pos[1]-1],retarr[pos[0]][pos[1]]
    return retarr
def move_up(state, pos):
    if(pos[0]==0):
        return None
    retarr = copy.deepcopy(state)
    #for i in state:
        #retarr.append(i)
    retarr[pos[0]][pos[1]],retarr[pos[0]-1][pos[1]] =
retarr[pos[0]-1][pos[1]],retarr[pos[0]][pos[1]]
    return retarr
def move_right(state, pos):
    if(pos[1]==2):
        return None
    retarr = copy.deepcopy(state)
    #for i in state:
        #retarr.append(i)
    retarr[pos[0]][pos[1]],retarr[pos[0]][pos[1]+1] =
retarr[pos[0]][pos[1]+1],retarr[pos[0]][pos[1]]
    return retarr
def move_down(state, pos):
    if(pos[0]==2):
        return None
    retarr = copy.deepcopy(state)
    retarr[pos[0]][pos[1]],retarr[pos[0]+1][pos[1]] =
retarr[pos[0]+1][pos[1]],retarr[pos[0]][pos[1]]
    return retarr
def printMatrix(matricesArray):
    print("")
    counter = 1
    for matrix in matricesArray:
        print("Step {}".format(counter))
        for row in matrix:
            print(row)
        counter+=1
        print("")
def eightPuzzle(initialstate, finalstate):
    hn=0
    explored = []
    while(True):
        explored.append(initialstate)
        if(initialstate == finalstate):
            break
        hn+=1
        left = move_left(initialstate, findposofblank(initialstate))
        right = move_right(initialstate, findposofblank(initialstate))
        up = move_up(initialstate, findposofblank(initialstate))
        down = move_down(initialstate, findposofblank(initialstate))
        fnl=1000; fnr=1000; fnu=1000;fnd=1000
        if(left!=None):
            fnl = hn + gn(left,finalstate)
        if(right!=None):
            fnr = hn + gn(right,finalstate)
        if(up!=None):
            fnu = hn + gn(up,finalstate)
        if(down!=None):
            fnd = hn + gn(down,finalstate)
        minfn = min(fnl, fnr, fnu, fnd)
        if((fnl == minfn) and (left not in explored)):
            initialstate = left
        elif((fnr == minfn) and (right not in explored)):
            initialstate = right
        elif((fnu == minfn) and (up not in explored)):
            initialstate = up
        elif((fnd == minfn) and (down not in explored)):
            initialstate = down
    printMatrix(explored)
#eightPuzzle(initial, final)
def main():
    while(True):
        ch = int(input("PRESS 1 to continue and 0 to Exit : "))
        if(not ch):
            break
        start = []
        print("START STATE\n")
        for i in range(3):
            arr=[]
            for j in range(3):
                a = int(input("Enter element at  {},{}: ".format(i,j)))
                arr.append(a)
            start.append(arr)
        final = []
        print("FINAL STATE\n")
        for i in range(3):
            arr=[]
            for j in range(3):
                a = int(input("Enter element at  {},{}: ".format(i,j)))
                arr.append(a)
            final.append(arr)
        eightPuzzle(start, final)
main()
```

OUTPUT:



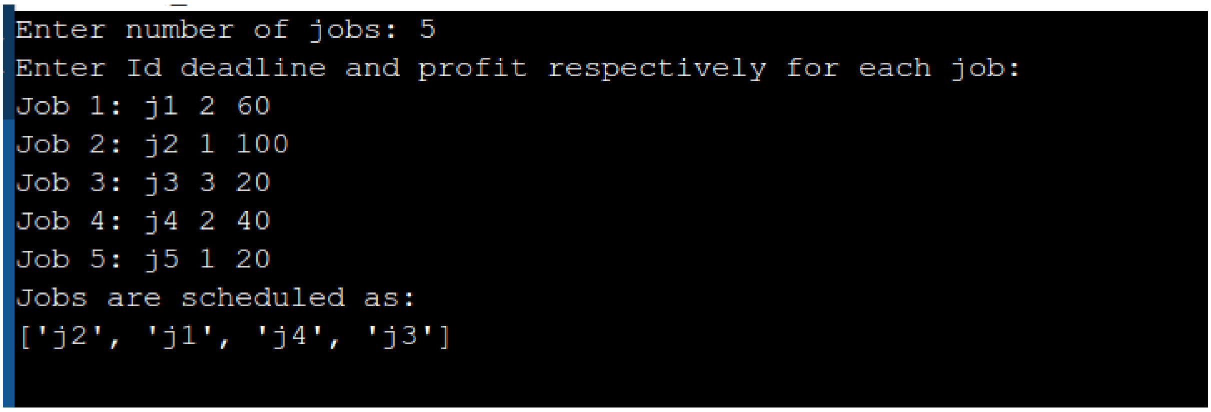
```
Enter element at  0,0: 1
Enter element at  0,1: 2
Enter element at  0,2: 3
Enter element at  1,0: -1
Enter element at  1,1: 4
Enter element at  1,2: 6
Enter element at  2,0: 7
Enter element at  2,1: 5
Enter element at  2,2: 8
FINAL STATE

Enter element at  0,0: 1
Enter element at  0,1: 2
Enter element at  0,2: 3
Enter element at  1,0: 4
Enter element at  1,1: 5
Enter element at  1,2: 6
Enter element at  2,0: 7
Enter element at  2,1: 8
Enter element at  2,2: -1

Step 1
[1, 2, 3]
[-1, 4, 6]
[7, 5, 8]

Step 2
[1, 2, 3]
[4, -1, 6]
[7, 5, 8]

Step 3
[1, 2, 3]
[4, 5, 6]
[7, -1, 8]

Step 4
[1, 2, 3]
[4, 5, 6]
[7, 8, -1]
```

ASSIGNMENT 03:

Job Scheduling Problem

```
n = int(input("Enter number of jobs: "))
jobs = []
print("Enter Id deadline and profit respectively for each job:")
for i in range(n):
    job = input("Job " + str(i+1) + ": ").split()
    jobs.append(job)

sorter = lambda job: int(job[2])
jobs = sorted(jobs, key=sorter, reverse=True)

scheduled = []
time = 0

for i in jobs:
    if time <= int(i[1]):
        scheduled.append(i[0])
        time += 1

print("Jobs are scheduled as: ")
print(scheduled)
```

OUTPUT:



```
Enter number of jobs: 5
Enter Id deadline and profit respectively for each job:
Job 1: j1 2 60
Job 2: j2 1 100
Job 3: j3 3 20
Job 4: j4 2 40
Job 5: j5 1 20
Jobs are scheduled as:
['j2', 'j1', 'j4', 'j3']
```

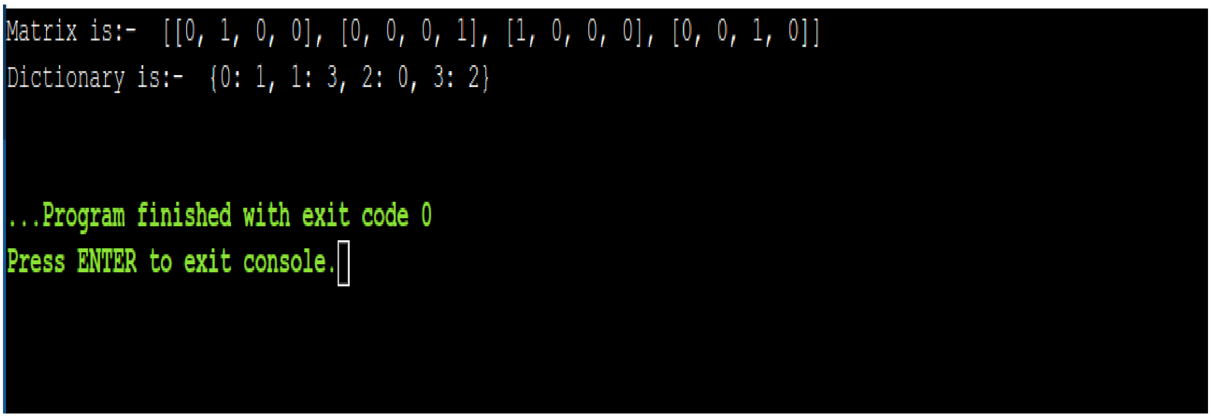
ASSIGNMENT 04:

This represents a solution to the 4-Queens problem, where there are four queens on a 4x4 chessboard and none of them attack each other. The matrix a represents the positions of the queens on the board, where a 1 represents a queen and a 0 represents an empty square. The dictionary b maps each row number to the column number where a queen is placed.

```
# Number of queens
n=4
# Matrix
a=[[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]]
# Dictionary for backtrack
b={}

# Checking if column is safe
def isColumnSafe(r,c):
    while(r>=0):
        if(a[r][c] == 1):
            return 0
        r = r-1
    return 1

# Checking if left diagonal is safe
def isLeftDiagonalSafe(r,c):
    while(r>=0 and c>=0):
        if(a[r][c] == 1):
            return 0
        r = r-1
        c = c-1
    return 1

# Checking if right diagonal is safe
def isRightDiagonalSafe(r,c):
    while(r>=0 and c<n):
        if(a[r][c]==1):
            return 0
        r = r-1
        c = c+1
    return 1
```

```
def isSafe(r,c):
    if(isColumnSafe(r,c) and isLeftDiagonalSafe(r,c) and
isRightDiagonalSafe(r,c)):
        return True
    return False
def chessboard(r,c):
    if(r>=n):
        return
    p = 0
    while c<n:
        p = isSafe(r,c)
        if p == 1:
            a[r][c] = 1
            b.update({r:c})
            break
        c=c+1

    if p==1:
        chessboard(r+1,0)
    else:
        a[r-1][b.get(r-1)]=0
        chessboard(r-1,int(b.get(r-1))+1)
chessboard(0,0)
print("Matrix is:- ",a)
print("Dictionary is:- ",b)
```

OUTPUT:

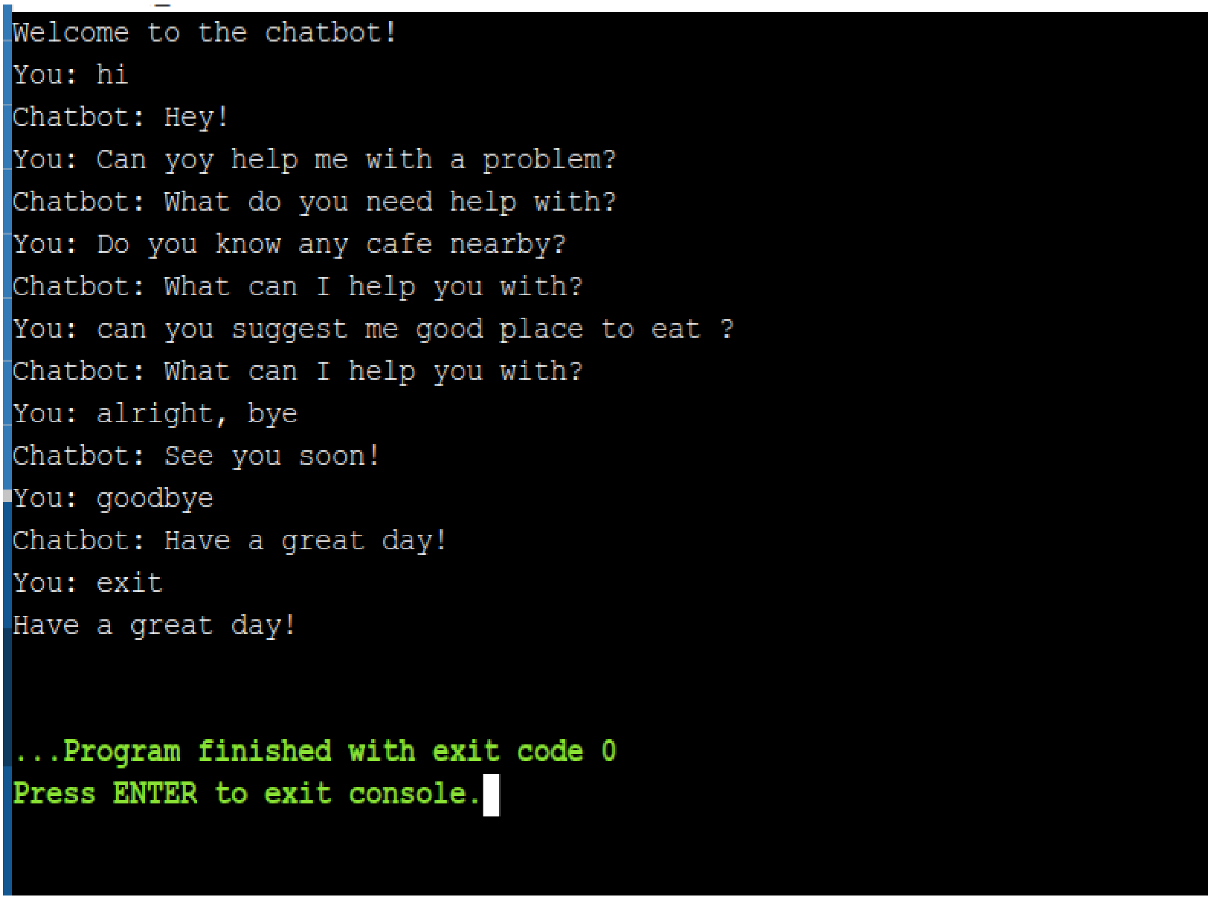


```
Matrix is:-  [[0, 1, 0, 0], [0, 0, 0, 1], [1, 0, 0, 0], [0, 0, 1, 0]]
Dictionary is:-  {0: 1, 1: 3, 2: 0, 3: 2}


...Program finished with exit code 0
Press ENTER to exit console.
```

ASSIGNMENT 05:

```
import random

# Define the chatbot's responses
greetings = ["Hello!", "Hi there!", "Hey!"]
questions = ["How can I assist you?", "What can I help you with?", "What do you need help with?"]
goodbyes = ["Goodbye!", "See you soon!", "Have a great day!"]
unknown = ["I'm sorry, I don't understand.", "Could you please rephrase that?", "I'm not sure what you mean."]

# Define a function to generate the chatbot's responses
def get_response(user_input):
    if "hello" in user_input.lower() or "hi" in user_input.lower() or "hey" in user_input.lower():
        return random.choice(greetings)
    elif "?" in user_input:
        return random.choice(questions)
    elif "bye" in user_input.lower() or "goodbye" in user_input.lower():
        return random.choice(goodbyes)
    else:
        return random.choice(unknown)

# Define the main function to interact with the user
def chat():
    print("Welcome to the chatbot!")
    while True:
        user_input = input("You: ")
        if user_input.lower() == "exit":
            print(random.choice(goodbyes))
            break
        else:
            bot_response = get_response(user_input)
            print("Chatbot:", bot_response)

# Call the main function to start the chat
chat()
```

OUTPUT:



```
Welcome to the chatbot!
You: hi
Chatbot: Hey!
You: Can yoy help me with a problem?
Chatbot: What do you need help with?
You: Do you know any cafe nearby?
Chatbot: What can I help you with?
You: can you suggest me good place to eat ?
Chatbot: What can I help you with?
You: alright, bye
Chatbot: See you soon!
You: goodbye
Chatbot: Have a great day!
You: exit
Have a great day!


...Program finished with exit code 0
Press ENTER to exit console.
```