

PRACTICAL-9

AIM: Design an XOR truth table using Python.

THEORY: The Exclusive-OR Gate/ XOR Gate is a combination of all the three basic gates (NOT, AND, OR gates). It receives two or more input signals but produces only one output signal. It results in a low '0' if the input bit pattern contains an even number of high '1' signals. If there are an odd number of high '1' signals, it results in high. For this reason, the Exclusive-OR Gate (XOR Gate) is called the Anti-Coincidence Gate or Inequality Detector or Odd Ones Detector.

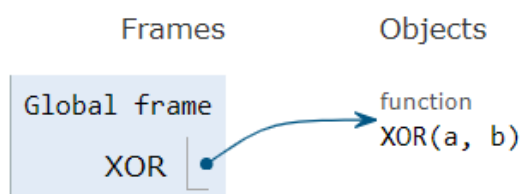
CODE:

```
def XOR (a, b):  
    if a != b:  
        return 1  
    else:  
        return 0  
  
# Driver code  
if __name__ == '__main__':  
    print(XOR(5, 5))  
    print("+-----+-----+")  
    print(" | XOR Truth Table | Result |")  
    print(" A = False, B = False | A XOR B =",XOR(False,False)," | ")  
    print(" A = False, B = True | A XOR B =",XOR(False,True)," | ")  
    print(" A = True, B = False | A XOR B =",XOR(True,False)," | ")  
    print(" A = True, B = True | A XOR B =",XOR(True,True)," | ")
```

OUTPUT:

Print output (drag lower right corner to resize)

```
| XOR Truth Table | Result |  
A = False, B = False | A XOR B = 0 |  
A = False, B = True | A XOR B = 1 |  
A = True, B = False | A XOR B = 1 |  
A = True, B = True | A XOR B = 0 |
```



Conclusion XOR table has been implemented using python.

Marking Criteria

Criteria	Total Marks	Marks Obtained	Comments
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6 (To be scaled down to 1.5)		

PRACTICAL-10

AIM: Study of SCIKIT fuzzy.

ABSTRACT

Scikit-Fuzzy is a versatile Python library for fuzzy logic operations, offering a wide range of functionalities for implementing fuzzy inference systems. This comprehensive study provides an in-depth overview of Scikit-Fuzzy, covering its installation process, documentation resources, and a detailed conclusion regarding its usage and significance in various applications. Through this study, readers will gain insights into installing Scikit-Fuzzy, navigating its documentation, and understanding its capabilities for addressing problems involving uncertainty and imprecision.

INTRODUCTION

Scikit-Fuzzy is a powerful Python library designed for fuzzy logic operations, providing a comprehensive set of tools for implementing fuzzy inference systems. It offers a user-friendly interface and a wide range of functionalities for defining fuzzy sets, creating fuzzy inference systems, and performing fuzzy logic operations such as fuzzification, rule evaluation, and defuzzification.

Scikit-Fuzzy simplifies the implementation of fuzzy logic systems by providing intuitive functions and classes for defining membership functions, fuzzy rules, and inference mechanisms. It supports various types of fuzzy logic systems, including Mamdani-type and Sugeno-type systems, making it suitable for a diverse range of applications.

One of the key advantages of Scikit-Fuzzy is its integration with the broader Python ecosystem, allowing users to leverage other libraries and frameworks for data manipulation, visualization, and machine learning. Additionally, Scikit-Fuzzy is open-source and actively maintained, with a growing community of users and developers contributing to its development and improvement.

INSTALLING SCIKIT-FUZZY

Obtain the source from the git-repository at <http://github.com/scikit-fuzzy/scikit-fuzzy> by running:

```
git clone http://github.com/scikit-fuzzy/scikit-fuzzy.git
```

in a terminal (you will need to have git installed on your machine).

If you do not have git installed, you can also download a zipball from <https://github.com/scikit-fuzzy/scikit-fuzzy/zipball/master>.

The SciKit can be installed globally using:

```
pip install -e .
```

or locally using:

```
python setup.py install --prefix=${HOME}
```

If you prefer, you can use it without installing, by simply adding this path to your PYTHONPATH variable.

While most functions are available in the base namespace, the package is factored with a logical grouping of functions in submodules. If the base namespace appears overwhelming, we recommend exploring them individually. These include

fuzz.membership

Fuzzy membership function generation

fuzz.defuzzify

Defuzzification algorithms to return crisp results from fuzzy sets

fuzz.fuzzymath

The core of scikit-fuzzy, containing the majority of the most common fuzzy logic operations.

fuzz.intervals

Interval mathematics. The restricted Dong, Shah, & Wong (DSW) methods for fuzzy set math live here.

fuzz.image

Limited fuzzy logic image processing operations.

fuzz.cluster

Fuzzy c-means clustering.

fuzz.filters

Fuzzy Inference Ruled by Else-action (FIRE) filters in 1D and 2D.

Fuzzy Control Primer

Fuzzy Control Primer Overview and Terminology Fuzzy Logic is a methodology predicated on the idea that the “truthiness” of something can be expressed over a continuum. This is to say that something isn’t true or false but instead partially true or partially false. A fuzzy variable has a crisp value which takes on some number over a pre-defined domain (in fuzzy logic terms, called a universe). The crisp value is how we think of the variable using normal mathematics. For example, if my fuzzy variable was how much to tip someone, it’s universe would be 0 to 25% and it might take on a crisp value of 15%. A fuzzy variable also has several terms that are used to describe the variable. The terms taken together are the fuzzy set which can be used to describe the “fuzzy value” of a fuzzy variable. These terms are usually adjectives like “poor,” “mediocre,” and “good.” Each term has a membership function that defines how a crisp value maps to the term on a scale of 0 to 1. In essence, it describes “how good” something is. So, back to the tip example, a “good tip” might have a membership function which has non-zero values between 15% and 25%, with 25% being a “completely good tip” (ie, it’s membership is 1.0) and 15% being a “barely good tip” (ie, its membership is 0.1). A fuzzy control system links fuzzy variables using a set of rules. These rules are simply mappings that describe how one or more fuzzy variables relates to another. These are expressed in terms of an IF-THEN statement; the IF part is called the antecedent and the THEN part is the consequent. In the tipping example, one rule might be “IF the service was good THEN the tip will be good.” The exact math related to how a rule is used to calculate the value of the consequent based on the value of the antecedent is outside the scope of this primer.

Marking Criteria

Criteria	Total Marks	Marks Obtained	Comments
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6 (To be scaled down to 1.5)		

PRACTICAL - 2

Aim- Write a program to implement Single Player Game

Tool Used- Vs Code

Theory- Snakes and ladder (Single player) Game

- There are 6 face dice which is being rolled by the player to their chance.
- The player starts from 0 and has to reach the final position (in our case, its 104).
- There are some ladder which turns out to be lucky for the player as they shorten the way.
- There are some snakes present in between the game which turns out to be the enemy of the player as they just lengthen their way to 104.

Code-

```
import random

import time

class SnakesAndLadder(object):

    def __init__(self, name, position):

        self.name = name

        self.position = position

        self.ladd = [4, 24, 48, 67, 86]

        self.lengthladd = [13, 23, 5, 12, 13]

        self.snake = [6, 26, 47, 23, 55, 97]

        self.lengthsnake = [4, 6, 7, 5, 8, 9]

    def dice(self):

        chances = 0

        print("-----Let's Start The Game-----\n")

        while self.position < 104:

            roll = random.choice([1, 2, 3, 4, 5, 6])

            print('Roll value:', roll)

            self.position += roll

            if self.position == 104:

                print('Completed the game')
```

```

        break

    if self.position in self.ladd:

        for n in range(len(self.ladd)):

            if self.position == self.ladd[n]:

                self.position += self.lengthladd[n]

                print('Climbed up a ladder!')

    if self.position in self.snake:

        for n in range(len(self.snake)):

            if self.position == self.snake[n]:

                self.position -= self.lengthsnake[n]

                print('Got bitten by a snake!')

    print('Current position of the player:', self.position, '\n')

    chances += 1

    time.sleep(1) # Add a delay for better readability

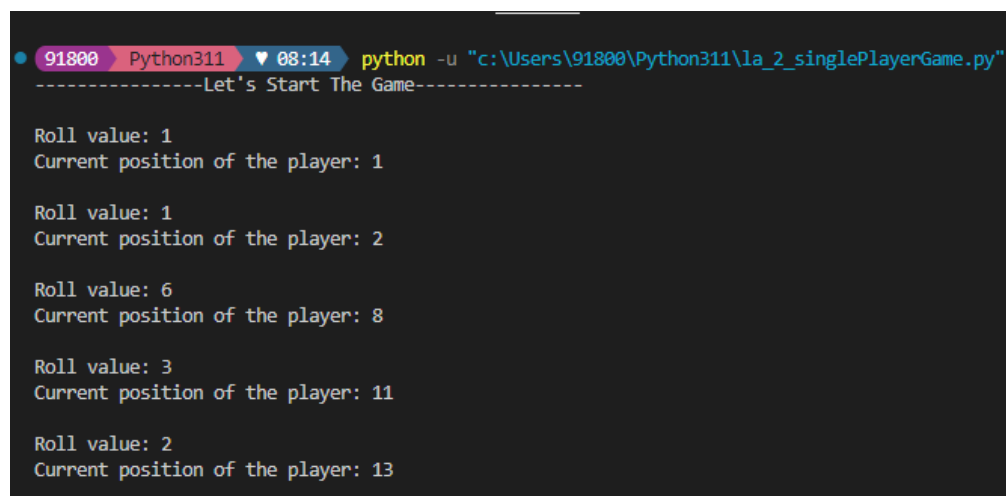
    print('Total number of chances:', chances)

zack = SnakesAndLadder('zack', 0)

zack.dice()

```

Output-



```

● 91800 Python311 ♥ 08:14 python -u "c:\Users\91800\Python311\la_2_singlePlayerGame.py"
-----Let's Start The Game-----

Roll value: 1
Current position of the player: 1

Roll value: 1
Current position of the player: 2

Roll value: 6
Current position of the player: 8

Roll value: 3
Current position of the player: 11

Roll value: 2
Current position of the player: 13

```

Conclusion: Single Player Game using Python is successfully implemented.

Marking Criteria

Criteria	Total Marks	Marks Obtained	Comments
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6 (To be scaled down to 1.5)		

PRACTICAL- 3

Aim- Write a program to implement the Tic-Tac-Toe game problem

Tool Used- Vs Code

Theory- Tic-Tac-Toe is among the games played between two players played on a **3 x 3 square grid**. Each player inhabits a cell in their respective turns, keeping the objective of placing three similar marks in a vertical, horizontal, or diagonal pattern. The first player utilizes the **Cross (X)** as the marker, whereas the other utilizes the **Naught or Zero (O)**.

Code-

```
import numpy as np
import random
from time import sleep

def create_board():
    return np.zeros((3, 3), dtype=int)

def possibilities(board):
    return [(i, j) for i in range(3) for j in range(3) if board[i][j] == 0]

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

def row_win(board, player):
    return any(all(cell == player for cell in row) for row in board)

def col_win(board, player):
    return any(all(board[row][col] == player for row in range(3)) for col in range(3))

def diag_win(board, player):
    return all(board[i][i] == player for i in range(3)) or all(board[i][2 - i] == player for i in range(3))

def evaluate(board):
    for player in [1, 2]:
        if row_win(board, player) or col_win(board, player) or diag_win(board, player):
            return player
    if np.all(board != 0):
        return -1
    return 0

def play_game():
    board, winner, counter = create_board(), 0, 1
```

```

print(board)

while winner == 0:
    for player in [1, 2]:
        board = random_place(board, player)
        print(f'Board after {counter} move:')
        print(board)
        sleep(1)
        counter += 1
        winner = evaluate(board)
        if winner != 0:
            break
        print("It's a tie!")
        return -1

    return winner

# Driver Code

result = play_game()

if result != -1:
    print(f'Player {result} wins!')

```

Output-

```

[[0 0 0]
 [0 0 0]
 [0 0 0]]
Board after 1 move:
[[0 0 0]
 [0 0 0]
 [1 0 0]]
Board after 2 move:
[[0 0 0]
 [0 0 0]
 [1 2 0]]
Board after 3 move:
[[0 0 0]
 [1 0 0]
 [1 2 0]]
Board after 4 move:
[[2 0 0]
 [1 0 0]
 [1 2 0]]
Board after 5 move:
[[2 0 0]
 [1 0 1]
 [1 2 0]]
Board after 6 move:
[[2 0 0]
 [1 0 1]
 [1 2 2]]
Board after 7 move:
[[2 0 1]
 [1 0 1]]

```

```

[[ 0  0  0]]
Board after 7 move:
[[2 0 1]
 [1 0 1]
 [1 2 2]]
Board after 8 move:
[[2 0 1]
 [1 2 1]
 [1 2 2]]
Player 2 wins!

```

Conclusion: Tic-Tac-Toe Game using Python is successfully implemented.

Marking Criteria

Criteria	Total Marks	Marks Obtained	Comments
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6 (To be scaled down to 1.5)		

PRACTICAL - 6

Aim: Write a program to implement BFS for water jug problem using Python

Theory: The water jug problem is a classic problem in computer science. It is a problem of finding a sequence of actions to transfer water between two jugs of different capacities such that a certain amount of water is reached in one of the jugs.

The problem can be solved using a variety of algorithms, including breadth-first search (BFS). BFS is a graph search algorithm that explores all possible states of the problem in a systematic way.

To solve the water jug problem using BFS, we start by creating a queue of states. The initial state is the state where both jugs are empty. We then repeatedly remove the first state from the queue and generate all possible next states. A next state is a state that can be reached from the current state by performing one of the following actions:

- Emptying one of the jugs
- Filling one of the jugs
- Pouring water from one jug to the other until one of the jugs is full or the other jug is empty

We then add each next state to the queue, if it has not already been visited. We continue this process until we find a state where the target amount of water is reached in one of the jugs.

If we reach a state where both jugs are empty, then the problem is unsolvable.

Code: from collections

```
import deque class Graph:
```

```
def __init__(self):
```

```
self.graph = {} def
```

```
add_edge(self, u, v): if u
```

```
not in self.graph:
```

```
self.graph[u] = []
```

```
self.graph[u].append(v)
```

```
def bfs(self, start_vertex):
```

```
    visited = set()
```

```
    queue = deque([start_vertex])
```

```
    while queue:
```

```

        vertex = queue.popleft()
    if vertex not in visited:
    print(vertex, end=' ')
    visited.add(vertex)        if
    vertex in self.graph:
        queue.extend(self.graph[vertex])

g = Graph()
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 2)
g.add_edge(2, 0)
g.add_edge(2, 3)
g.add_edge(3, 3)

print("BFS starting from vertex 2:")
g.bfs(2)

```

Output:



```

91800 Python311 08:22 python -u "c:\Users\91800\Python311\AI_bfs.py"
BFS starting from vertex 2:
2 0 3 1
91800 Python311 08:24

```

Conclusion: BFS for water jug problem using Python is successfully implemented.

Marking Criteria

Criteria	Total Marks	Marks Obtained	Comments
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		

Total	6 (To be scaled down to 1.5)
-------	------------------------------

PRACTICAL-7

Aim- Write a Program to implement DFS using Python

Theory - Depth First Search (DFS) is an algorithm that explores a graph or a tree by going as deep as possible along each branch before backtracking. It uses a stack data structure to keep track of the visited and unvisited nodes. DFS can be implemented recursively or iteratively in Python.

```
Code- class
Graph:  def
__init__(self):
self.graph = {}

    def add_edge(self, u, v):
if u not in self.graph:
self.graph[u] = []
    self.graph[u].append(v)

    def dfs(self, start_vertex, visited=None):
if visited is None:
    visited = set()

    if start_vertex not in visited:
print(start_vertex, end=' ')
    visited.add(start_vertex)

    if start_vertex in self.graph:
        for
neighbor in self.graph[start_vertex]:
            self.dfs(neighbor, visited)

# Example usage: g
= Graph()
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 2)
g.add_edge(2, 0)
g.add_edge(2, 3)
g.add_edge(3, 3)
```

```
print("DFS starting from vertex 2:") g.dfs(2)
```

Output:-

```
91800 Python311 08:24 python -u "c:\Users\91800\Python311\AI_bfs.py"
DFS starting from vertex 2:
2 0 1 3
91800 Python311 08:29
```

Conclusion: DFS using Python is successfully implemented.

Marking Criteria

Criteria	Total Marks	Marks Obtained	Comments
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6 (To be scaled down to 1.5)		

PRACTICAL - 2

Aim- Write a program to implement Single Player Game

Tool Used- Vs Code

Theory- Snakes and ladder (Single player) Game

- There are 6 face dice which is being rolled by the player to their chance.
- The player starts from 0 and has to reach the final position (in our case, its 104).
- There are some ladder which turns out to be lucky for the player as they shorten the way.
- There are some snakes present in between the game which turns out to be the enemy of the player as they just lengthen their way to 104.

Code-

```
import random

import time

class SnakesAndLadder(object):

    def __init__(self, name, position):

        self.name = name

        self.position = position

        self.ladd = [4, 24, 48, 67, 86]

        self.lengthladd = [13, 23, 5, 12, 13]

        self.snake = [6, 26, 47, 23, 55, 97]

        self.lengthsnake = [4, 6, 7, 5, 8, 9]

    def dice(self):

        chances = 0

        print("-----Let's Start The Game-----\n")

        while self.position < 104:

            roll = random.choice([1, 2, 3, 4, 5, 6])

            print('Roll value:', roll)

            self.position += roll

            if self.position == 104:

                print('Completed the game')
```



```

        break

    if self.position in self.ladd:

        for n in range(len(self.ladd)):

            if self.position == self.ladd[n]:

                self.position += self.lengthladd[n]

                print('Climbed up a ladder!')

    if self.position in self.snake:

        for n in range(len(self.snake)):

            if self.position == self.snake[n]:

                self.position -= self.lengthsnake[n]

                print('Got bitten by a snake!')

    print('Current position of the player:', self.position, '\n')

    chances += 1

    time.sleep(1) # Add a delay for better readability

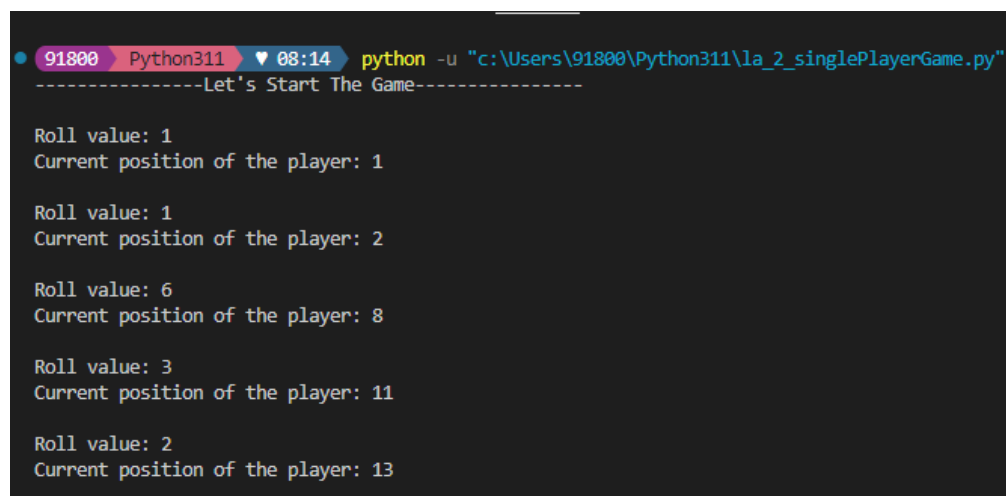
    print('Total number of chances:', chances)

zack = SnakesAndLadder('zack', 0)

zack.dice()

```

Output-



```

● 91800 Python311 ♥ 08:14 python -u "c:\Users\91800\Python311\la_2_singlePlayerGame.py"
-----Let's Start The Game-----

Roll value: 1
Current position of the player: 1

Roll value: 1
Current position of the player: 2

Roll value: 6
Current position of the player: 8

Roll value: 3
Current position of the player: 11

Roll value: 2
Current position of the player: 13

```

Conclusion: Single Player Game using Python is successfully implemented.

Marking Criteria

Criteria	Total Marks	Marks Obtained	Comments
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6 (To be scaled down to 1.5)		

PRACTICAL- 3

Aim- Write a program to implement the Tic-Tac-Toe game problem

Tool Used- Vs Code

Theory- Tic-Tac-Toe is among the games played between two players played on a **3 x 3 square grid**. Each player inhabits a cell in their respective turns, keeping the objective of placing three similar marks in a vertical, horizontal, or diagonal pattern. The first player utilizes the **Cross (X)** as the marker, whereas the other utilizes the **Naught or Zero (O)**.

Code-

```
import numpy as np
import random
from time import sleep

def create_board():
    return np.zeros((3, 3), dtype=int)

def possibilities(board):
    return [(i, j) for i in range(3) for j in range(3) if board[i][j] == 0]

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

def row_win(board, player):
    return any(all(cell == player for cell in row) for row in board)

def col_win(board, player):
    return any(all(board[row][col] == player for row in range(3)) for col in range(3))

def diag_win(board, player):
    return all(board[i][i] == player for i in range(3)) or all(board[i][2 - i] == player for i in range(3))

def evaluate(board):
    for player in [1, 2]:
        if row_win(board, player) or col_win(board, player) or diag_win(board, player):
            return player
    if np.all(board != 0):
        return -1
    return 0

def play_game():
    board, winner, counter = create_board(), 0, 1
```

```

print(board)

while winner == 0:
    for player in [1, 2]:
        board = random_place(board, player)
        print(f'Board after {counter} move:')
        print(board)
        sleep(1)
        counter += 1
        winner = evaluate(board)
        if winner != 0:
            break
        print("It's a tie!")
        return -1

    return winner

# Driver Code

result = play_game()

if result != -1:
    print(f'Player {result} wins!')

```

Output-

```

[[0 0 0]
 [0 0 0]
 [0 0 0]]
Board after 1 move:
[[0 0 0]
 [0 0 0]
 [1 0 0]]
Board after 2 move:
[[0 0 0]
 [0 0 0]
 [1 2 0]]
Board after 3 move:
[[0 0 0]
 [1 0 0]
 [1 2 0]]
Board after 4 move:
[[2 0 0]
 [1 0 0]
 [1 2 0]]
Board after 5 move:
[[2 0 0]
 [1 0 1]
 [1 2 0]]
Board after 6 move:
[[2 0 0]
 [1 0 1]
 [1 2 2]]
Board after 7 move:
[[2 0 1]
 [1 0 1]]

```

```

[[ 0  0  0]]
Board after 7 move:
[[2 0 1]
 [1 0 1]
 [1 2 2]]
Board after 8 move:
[[2 0 1]
 [1 2 1]
 [1 2 2]]
Player 2 wins!

```

Conclusion: Tic-Tac-Toe Game using Python is successfully implemented.

Marking Criteria

Criteria	Total Marks	Marks Obtained	Comments
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6 (To be scaled down to 1.5)		

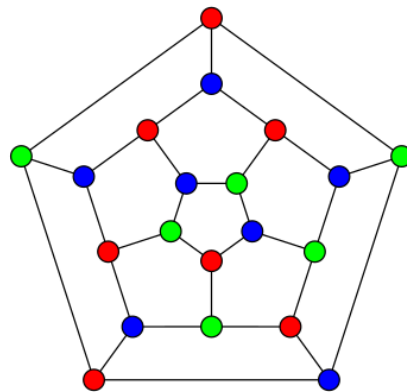
PRACTICAL – 5

Aim- Implement Graph coloring problem using Python.

Tool Used- VS Code

Theory- The graph coloring problem is a fundamental puzzle in graph theory where the aim is to color the vertices of a graph in such a way that no two adjacent vertices share the same color. This problem finds applications in various fields like scheduling, register allocation in compilers, and solving Sudoku puzzles. Formally, given an undirected graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, the objective is to find a coloring function $c: V \rightarrow C$, where C is a set of colors, such that $c(u) \neq c(v)$ for every edge (u, v) in E . The time complexity is given below, where m is the number of colors in the graph and V is the number of vertices.

Time Complexity (Brute) - $O(m^V)$



Code-

```
def addEdge(adj, v, w):
    adj[v].append(w)
    # the graph is undirected
    adj[w].append(v)
    return adj

def greedyColoring(adj, V):

    result = [-1] * V
    result[0] = 0
    available = [False] * V

    # Assign colors to remaining V-1 vertices
    for u in range(1, V):
        for i in adj[u]:
            if (result[i] != -1):
                available[result[i]] = True
```

```

# Find the first available color

cr = 0

while cr < V:
    if (available[cr] == False):
        break

    cr += 1

# Assign the found color
result[u] = cr

for i in adj[u]:
    if (result[i] != -1):
        available[result[i]] = False

# Print the result
for u in range(V):
    print("Vertex", u, " ---> Color", result[u])

# Driver Code
if __name__ == '__main__':

    g1 = [[] for i in range(5)]
    g1 = addEdge(g1, 0, 1)
    g1 = addEdge(g1, 0, 2)
    g1 = addEdge(g1, 1, 2)
    g1 = addEdge(g1, 1, 3)
    g1 = addEdge(g1, 2, 3)
    g1 = addEdge(g1, 3, 4)
    print("Coloring of graph 1 ")
    greedyColoring(g1, 5)

    g2 = [[] for i in range(5)]
    g2 = addEdge(g2, 0, 1)
    g2 = addEdge(g2, 0, 2)
    g2 = addEdge(g2, 1, 2)
    g2 = addEdge(g2, 1, 4)
    g2 = addEdge(g2, 2, 4)
    g2 = addEdge(g2, 4, 3)

```

```
print("\nColoring of graph 2")
greedyColoring(g2, 5)
```

Output-

```
Vertex 4 ---> Color 3
91800 Python311 08:02 python -u "c:\Users\91800\Python311\graph_coloring.py"
Coloring of graph 1
Vertex 0 ---> Color 0
Vertex 1 ---> Color 1
Vertex 2 ---> Color 2
Vertex 3 ---> Color 0
Vertex 4 ---> Color 1

Coloring of graph 2
Vertex 0 ---> Color 0
Vertex 1 ---> Color 1
Vertex 2 ---> Color 2
Vertex 3 ---> Color 0
Vertex 4 ---> Color 3
91800 Python311 08:03
```

Marking Criteria

Criteria	Total Marks	Marks Obtained	Comments
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6 (To be scaled down to 1.5)		

PRACTICAL - 6

Aim: Write a program to implement BFS for water jug problem using Python

Theory: The water jug problem is a classic problem in computer science. It is a problem of finding a sequence of actions to transfer water between two jugs of different capacities such that a certain amount of water is reached in one of the jugs.

The problem can be solved using a variety of algorithms, including breadth-first search (BFS). BFS is a graph search algorithm that explores all possible states of the problem in a systematic way.

To solve the water jug problem using BFS, we start by creating a queue of states. The initial state is the state where both jugs are empty. We then repeatedly remove the first state from the queue and generate all possible next states. A next state is a state that can be reached from the current state by performing one of the following actions:

- Emptying one of the jugs
- Filling one of the jugs
- Pouring water from one jug to the other until one of the jugs is full or the other jug is empty

We then add each next state to the queue, if it has not already been visited. We continue this process until we find a state where the target amount of water is reached in one of the jugs.

If we reach a state where both jugs are empty, then the problem is unsolvable.

Code: from collections

```
import deque class Graph:
```

```
def __init__(self):
```

```
self.graph = {} def
```

```
add_edge(self, u, v): if u
```

```
not in self.graph:
```

```
self.graph[u] = []
```

```
self.graph[u].append(v)
```

```
def bfs(self, start_vertex):
```

```
visited = set()
```

```
queue = deque([start_vertex])
```

```
while queue:
```

```
vertex = queue.popleft()
```

```
if vertex not in visited:
```

```

print(vertex, end=' ')
visited.add(vertex)          if
vertex in self.graph:
    queue.extend(self.graph[vertex])

```

```

g = Graph()
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 2)
g.add_edge(2, 0)
g.add_edge(2, 3)
g.add_edge(3, 3)

```

```

print("BFS starting from vertex 2:")
g.bfs(2)

```

Output:

```

python -u "c:\Users\91800\Python311\AI_bfs.py"
BFS starting from vertex 2:
2 0 3 1

```

Conclusion: BFS for water jug problem using Python is successfully implemented.

Marking Criteria

Criteria	Total Marks	Marks Obtained	Comments
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6 (To be scaled down to 1.5)		

PRACTICAL-7

Aim- Write a Program to implement DFS using Python

Theory - Depth First Search (DFS) is an algorithm that explores a graph or a tree by going as deep as possible along each branch before backtracking. It uses a stack data structure to keep track of the visited and unvisited nodes. DFS can be implemented recursively or iteratively in Python.

```
Code- class
Graph: def
__init__(self):
self.graph = {}

def add_edge(self, u, v):
if u not in self.graph:
self.graph[u] = []
self.graph[u].append(v)

def dfs(self, start_vertex, visited=None):
if visited is None:
visited = set()

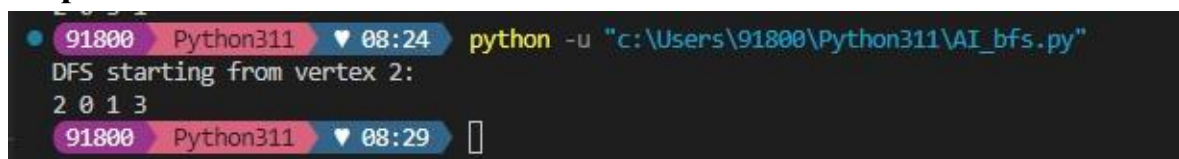
if start_vertex not in visited:
print(start_vertex, end=' ')
visited.add(start_vertex)

if start_vertex in self.graph:
for
neighbor in self.graph[start_vertex]:
self.dfs(neighbor, visited)

# Example usage: g
= Graph()
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 2)
g.add_edge(2, 0)
g.add_edge(2, 3)
g.add_edge(3, 3)
```

```
print("DFS starting from vertex 2:") g.dfs(2)
```

Output:-



```
91800 Python311 ▼ 08:24 python -u "c:\Users\91800\Python311\AI_bfs.py"
DFS starting from vertex 2:
2 0 1 3
91800 Python311 ▼ 08:29 []
```

Conclusion: DFS using Python is successfully implemented.

Marking Criteria

Criteria	Total Marks	Marks Obtained	Comments
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6 (To be scaled down to 1.5)		

PRACTICAL-4

Aim- Implement Brute Force solution to the knapsack problem in Python.

Tool used- VS Code

Theory- Given **N** items where each item has some weight and profit associated with it and also given a bag with capacity **W**, [i.e., the bag can hold at most **W** weight in it]. The task is to put the items into the bag such that the sum of profits associated with them is the maximum possible.

Here knapsack is like a container or a bag. Suppose we have given some items which have some weights or profits. We have to put some items in the knapsack in such a way total value produces a maximum profit. For example, the weight of the container is 20 kg. We have to select the items in such a way that the sum of the weight of items should be either smaller than or equal to the weight of the container, and the profit should be maximum.

There are two types of knapsack problems:

- 0/1 knapsack problem
- Fractional knapsack problem

Code-

```
def knapSack(W, wt, val, n)
    K = [[0 for x in range(W + 1)] for x in range(n + 1)]
    for i in range(n + 1):
        for w in range(W + 1):
            if i == 0 or w == 0:
                K[i][w] = 0
            elif wt[i-1] <= w:
                K[i][w] = max(val[i-1]
                               + K[i-1][w-wt[i-1]],
                               K[i-1][w])
            else:
                K[i][w] = K[i-1][w]
```

```

        return K[n][W]
if __name__ == '__main__':
    profit = [60, 100, 120]
    weight = [10, 20, 30]
    W = 50
    n = len(profit)
    print(knapSack(W, weight, profit, n))

```

Result-

```

220

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

```

Marking Criteria for Experiment-4

Criteria	Total Marks	Marks Obtained	Comments
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6 (To be scaled down to 1.5)		

PRACTICAL-8

AIM: Tokenization of a sentence with NLTK Package in python

THEORY: The NLTK (Natural Language Toolkit) package in Python is a powerful library for natural language processing. Tokenization is the process of breaking down a text into individual words or tokens. From rudimentary tasks such as text pre-processing to tasks like vectorized representation of text – NLTK's API has covered everything. NLTK is Python's API library for performing an array of tasks in human language. It can perform a variety of operations on textual data, such as classification, tokenization, stemming, tagging, Le parsing, semantic reasoning, etc.

CODE:

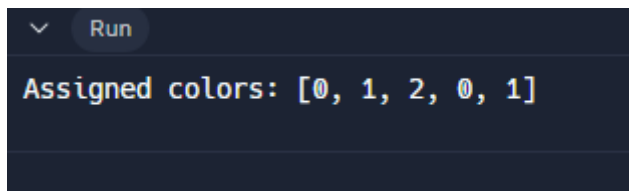
```
import nltk

sentence = "This is a sample sentence for tokenization using NLTK in Python."

tokens = nltk.word_tokenize(sentence)

print(tokens)
```

OUTPUT:



```
Assigned colors: [0, 1, 2, 0, 1]
```

Conclusion: Tokenization of a sentence with NLTK Package in python was done successfully.

Marking Criteria

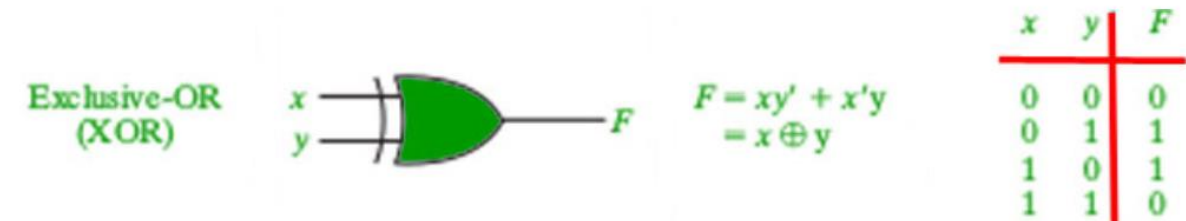
Criteria	Total Marks	Marks Obtained	Comments
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6 (To be scaled down to 1.5)		

PRACTICAL – 9

Aim- Design an XOR truth table using Python.

Tool Used- Visual Studio Code

Theory- The XOR gate gives an output of 1 if either of the inputs is different, it gives 0 if they are the same.



Code-

```
def XOR(a, b):  
    if a != b:  
        return 1  
    else:  
        return 0  
  
if __name__ == '__main__':  
    print(XOR(5, 5))  
  
    print("+-----+-----+")  
    print("| XOR Truth Table | Result |")  
  
    print(" A = False, B = False | A XOR B =", XOR(False, False), " | ")  
    print(" A = False, B = True | A XOR B =", XOR(False, True), " | ")  
    print(" A = True, B = False | A XOR B =", XOR(True, False), " | ")  
    print(" A = True, B = True | A XOR B =", XOR(True, True), " | ")
```

Output-

```
0  
+-----+-----+  
| XOR Truth Table | Result |  
A = False, B = False | A XOR B = 0 |  
A = False, B = True | A XOR B = 1 |  
A = True, B = False | A XOR B = 1 |  
A = True, B = True | A XOR B = 0 |
```


Marking Criteria

Criteria	Total Marks	Marks Obtained	Comments
Concept (A)	2		
Implementation (B)	2		
Performance (C)	2		
Total	6 (To be scaled down to 1.5)		