EXPERIMENT 1

INTRODUCTION OF VARIOUS PYTHON LIBRARIES USED FOR ARTIFICIAL INTELLIGENCE

AIM: To study the various Libraries used for Artificial Intelligence.

OJECTIVES:

1. To explore the ecosystem of Python libraries used in AI development.
2. To understand how libraries support AI tasks such as data handling, visualization, and learning.
3. To learn how to install, import, and use these libraries in practical AI projects.
4. To relate each library's role to robotics applications.

THEORY:

1. Importance of Python in Artificial Intelligence

Python has become the dominant language for AI and machine learning due to its simplicity, readability, and vast collection of libraries. AI projects involve multiple steps data collection, preprocessing, training models, visualization, and deployment and Python provides specialized libraries for each of these steps.

In robotics, AI libraries enable perception, reasoning, and learning allowing robots to navigate, plan, and interact with their environments intelligently.

2. Overview of Major AI Libraries

a) NumPy (Numerical Python)

NumPy provides a foundation for numerical computing. It introduces the ndarray (n-dimensional array), which is more efficient than Python lists for handling large data.

- **Applications:**
  - Mathematical operations and matrix manipulation.
  - Vectorized computations for faster performance.
  - Foundation for deep learning tensor operations.
- **Robotics Use:**
  Processing sensor arrays, control data, and robotic arm kinematics.

b) Pandas

Pandas is used for data manipulation, cleaning, and analysis. It introduces Series (1D) and DataFrame (2D) data structures.

- **Applications:**
    - Data cleaning and handling missing values.
    - Statistical analysis of datasets.
    - Organizing sensor or performance logs.
- **Robotics Use:**
  Managing large amounts of robot performance data or experiment results.

c) Matplotlib and Seaborn

These are the most common data visualization libraries in Python. Matplotlib creates static plots such as line graphs, bar charts, and scatter plots. Seaborn adds advanced statistical graphics with minimal code.

Applications:

- Visualizing AI model accuracy and performance metrics.
- Displaying sensor readings, control trajectories, and data trends.

Robotics Use:
Plotting motion trajectories, sensor behavior, or robot arm movement over time.

d) Scikit-learn

A comprehensive machine learning library providing ready-to-use implementations of classical algorithms.

- **Features:**
    - Classification (SVM, Decision Trees, Naïve Bayes).
    - Regression (Linear, Ridge, Polynomial).
    - Clustering (K-Means, Hierarchical).
    - Model evaluation and data preprocessing.
- **Robotics Use:**
  Behavior prediction, environment mapping, fault detection.

e) TensorFlow and PyTorch

These are deep learning frameworks that enable complex neural network modelling.

TensorFlow:

- Developed by Google.
- Uses computational graphs for efficient large-scale machine learning.
- Supports distributed training and deployment on CPUs, GPUs, and TPUs.

PyTorch:

- Developed by Meta (Facebook).
- Known for dynamic computation graphs and user-friendly syntax.
- Favored in research and robotics due to flexibility.

Applications:

- Object detection and recognition.
- Speech and gesture recognition.
- Reinforcement learning in autonomous robots.

f) OpenCV (Open-Source Computer Vision Library)

A real-time computer vision library for image and video analysis.

Applications:

- Image processing (filtering, segmentation).
- Object detection and tracking.
- Camera calibration and motion analysis.

Robotics Use:
Enabling robots to perceive their environment for obstacle avoidance, localization, and manipulation.

g) NLTK and spaCy

Libraries for Natural Language Processing (NLP).

- **Applications:**
  - Text classification, tokenization, and named entity recognition.
  - Building chatbots and voice command systems.

- **Robotics Use:**
  Voice-command-based robot control and human-robot interaction.

h) Other Libraries

- **Keras:** High-level neural network API that runs on TensorFlow.
- **Gym (OpenAI):** Toolkit for reinforcement learning research.
- **SymPy:** Symbolic mathematics and equation solving.

CONCLUSION:

Python’s diverse AI libraries empower developers to build intelligent systems across perception, reasoning, and learning. In robotics, these libraries form the computational core for autonomous and adaptive robot behavior.

EXPERIMENT 2

INTRODUCTION TO FUNDAMENTALS OF ARTIFICIAL INTELLIGENCE SUCH AS AGENTS, STACK, QUEUE, TREE, GRAPH, AND SEARCH CONCEPTS

AIM: To understand the fundamental concepts and building blocks of Artificial Intelligence including intelligent agents, data structures (stack, queue, tree, graph), and search algorithms used in problem-solving.

OBJECTIVES:

1. To understand what intelligence means in the context of machines.
2. To study the structure and types of intelligent agents.
3. To understand basic data structures supporting AI problem-solving.
4. To explore various search techniques used in AI decision-making.

THEORY:

1. What is Artificial Intelligence?

Artificial Intelligence (AI) is the field of computer science that focuses on creating machines capable of performing tasks that typically require human intelligence such as perception, reasoning, learning, problem-solving, and decision-making.

AI systems are designed to:

- Perceive their environment through sensors.
- Process and represent knowledge.
- Act intelligently to achieve goals.
- Learn and improve over time.

AI integrates subfields such as machine learning, computer vision, robotics, natural language processing, and expert systems.

2. Intelligent Agents

An agent is anything that can perceive its environment and act upon it.

Agent Function:
A mapping from precepts (inputs) to actions (outputs).
Formally: Agent Function = f(Percept) → Action

Components of an Agent:

- **Sensors:** Perceive the environment.
- **Actuators:** Perform actions.
- **Percept Sequence:** Complete history of what the agent has perceived.
- **Agent Program:** Implements the mapping from percepts to actions.

Types of Agents:

1. **Simple Reflex Agents:**
   React directly to current precepts. Example: A robot stopping when an obstacle is detected.
2. **Model-based Reflex Agents:**
   Maintain an internal model of the world.
3. **Goal-based Agents:**
   Select actions to achieve specific goals.
4. **Utility-based Agents:**
   Optimize performance by measuring utility.
5. **Learning Agents:**
   Improve behaviour using experience.

3. Data Structures in AI

AI problems often involve navigating state spaces, where each state represents a configuration of the world.
To represent and manipulate these states, various data structures are used:

- **Stack (LIFO):** Used for Depth-First Search (DFS) algorithms.
- **Queue (FIFO):** Used for Breadth-First Search (BFS) algorithms.
- **Tree:** Represents hierarchical relationships such as decision trees or game trees.
- **Graph:** Represents connected networks like state spaces or pathfinding problems.

These structures are foundational to AI’s ability to reason and plan.

4. Search in Artificial Intelligence

Search is the process of exploring possible actions and states to reach a goal. It is central to AI problem-solving.

Types of Searches:

- **Uninformed (Blind) Search:**
  No additional knowledge about the goal other than the problem definition.
  Examples:
  - Breadth-First Search (BFS)
  - Depth-First Search (DFS)
  - Uniform Cost Search
- **Informed (Heuristic) Search:**
  Uses problem-specific knowledge (heuristics) to guide the search.
  Examples:
  - A* Algorithm
  - Greedy Best-First Search

Key Search Components:

- **Initial State:** Where the search begins.
- **Goal State:** Desired target.
- **Successor Function:** Defines possible actions.
- **Path Cost:** Evaluates the efficiency of paths.

5. Applications of AI in Robotics

AI allows robots to function autonomously by:

- **Perception:** Using sensors and computer vision to understand surroundings.
- **Decision Making:** Planning optimal actions using search and reasoning.
- **Learning:** Improving performance through machine learning.
- **Interaction:** Communicating with humans using natural language or gestures.

CONCLUSION:

Understanding AI fundamentals agents, structures, and search forms the conceptual backbone for building intelligent robots. These principles enable robots to make decisions, learn, and interact intelligently in dynamic environments.

EXPERIMENT 3

IMPLEMENT BREADTH FIRST SEARCH USING PYTHON

AIM: To implement the Breadth-First Search algorithm in Python for exploring or traversing a graph**.**

OBJECTIVES:

1. To understand the concept of breadth-first traversal.
2. To implement BFS using a queue data structure.
3. To observe the order of node exploration in an unweighted graph.

THEORY:

Breadth-First Search (BFS) is an uninformed search algorithm that explores all neighbouring nodes at the current depth level before moving on to nodes at the next level.
It uses a queue (FIFO) to keep track of the nodes to be visited.

Applications:

- Shortest path in unweighted graphs.
- Network routing and social network analysis.
- Finding connected components in graphs.

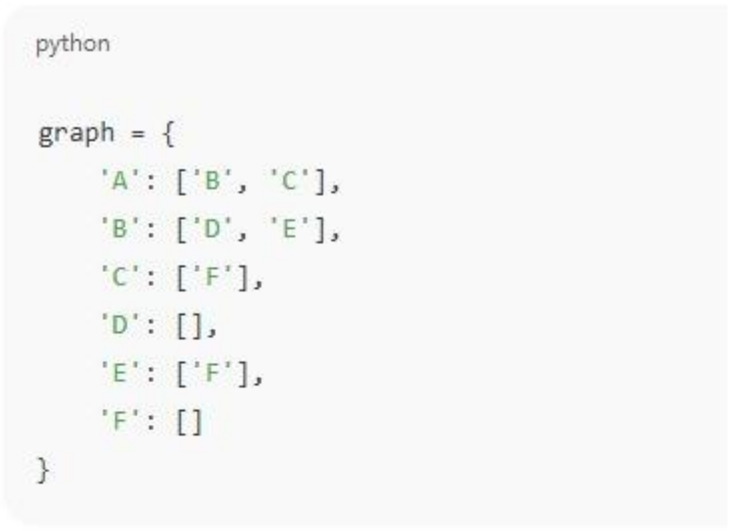
Algorithm:

1. Start from the initial node and mark it as visited.
2. Insert the node into a queue.
3. Repeat the following steps until the queue becomes empty:
   a. Dequeue a node from the front of the queue.
   b. Visit all unvisited adjacent nodes of the current node.
   c. Mark them as visited and enqueue them.
4. Continue until all reachable nodes are visited.

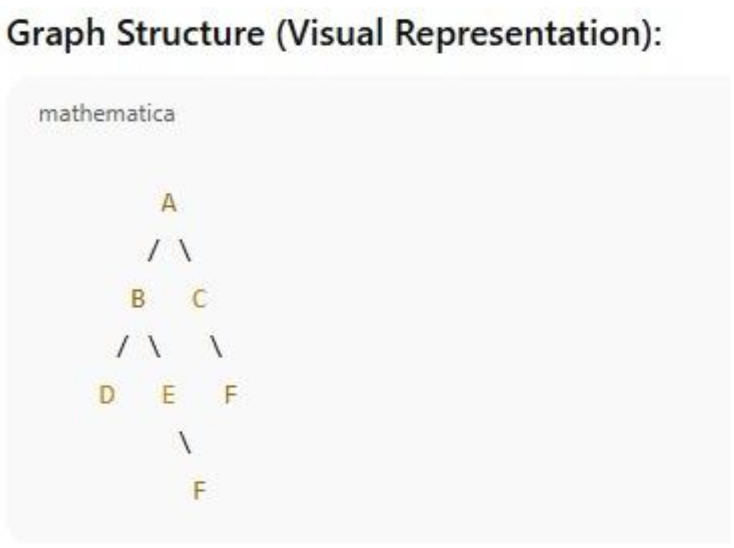
CODE:

```
from collections import deque
```

```
def bfs (graph, start):
    visited = set ()
    queue = deque([start])
    while queue:
        node = queue.popleft()
        if node not in visited:
            print(node, end=" ")
            visited.add(node)
            queue.extend(neighbor for neighbor in graph[node] if neighbor not in visited)

# Example graph represented as an adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}

print("Breadth First Search traversal:")
bfs(graph, 'A')
```


python

```python
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}
```


Graph Structure (Visual Representation):

mathematica

```
    A
   / \
  B   C
 / \   \
D   E   F
     \
      F
```

OUTPUT:

Breadth First Search traversal:

A B C D E F

CONCLUSION:

BFS explores nodes level by level using a queue, making it suitable for finding the shortest path in unweighted graphs.

EXPERIMENT 4

IMPLEMENT DEPTH-FIRST SEARCH (DFS) USING PYTHON

AIM: To implement the Depth-First Search algorithm in Python for graph traversal.

OBJECTIVES:

1. To understand the depth-first traversal mechanism.
2. To implement DFS using recursion or a stack.
3. To analyze traversal order in graphs.

THEORY:
Depth-First Search (DFS) is an uninformed search algorithm that explores as far as possible along a branch before backtracking.
It uses a stack (LIFO), which can be implemented either explicitly or through recursion.

Applications:

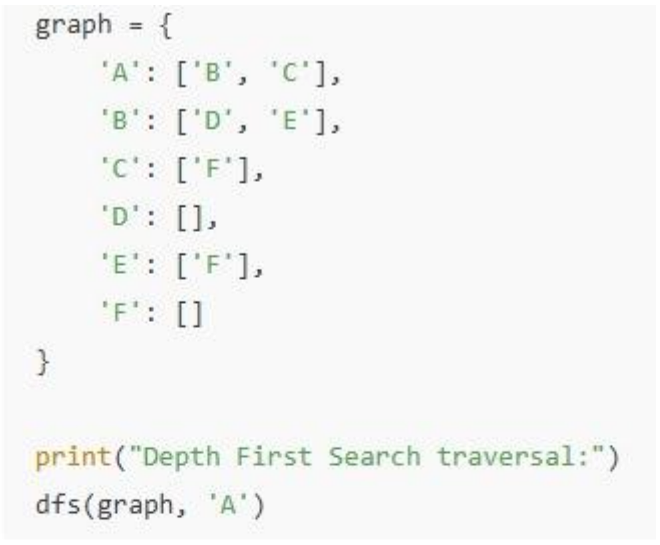
- Topological sorting.
- Pathfinding in mazes.
- Detecting cycles in graphs.

Algorithm:

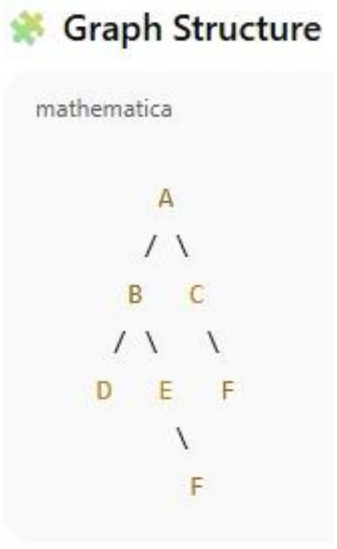
1. Start from the initial node and mark it as visited.
2. Explore any unvisited adjacent node recursively.
3. If no adjacent unvisited node exists, backtrack to the previous node.
4. Repeat until all nodes are visited.

CODE:

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start, end=" ")
    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)

# Example graph
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}

print("Depth First Search traversal:")
dfs(graph, 'A')
```



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

print("Depth First Search traversal:")
dfs(graph, 'A')
```



Graph Structure

mathematica

```
      A
     / \
    B   C
   / \   \
  D   E   F
       \
        F
```

OUTPUT:

Depth First Search traversal:

```
A B D E F
```

CONCLUSION:
DFS explores paths deeply before backtracking, making it useful for exhaustive search and problem-solving in recursive structures.

EXPERIMENT 5

IMPLEMENT A SEARCH USING PYTHON

AIM:
To implement the A* search algorithm in Python for finding the optimal path in a weighted graph.

OBJECTIVES:

1. To understand the working principle of heuristic-based search.
2. To implement A* for shortest-path estimation.
3. To study the role of cost function and heuristics.

THEORY:
A* (A-star) is an informed search algorithm that uses both path cost (g) and heuristic estimate (h) to determine the best next node to explore.

The evaluation function is:

$$f(n) = g(n) + h(n)$$

- g(n): Cost from start to current node.
- h(n): Estimated cost from current node to goal.
- f(n): Total estimated cost of the solution path.

Applications:

- Pathfinding in robotics and video games.
- Route optimization in navigation systems.

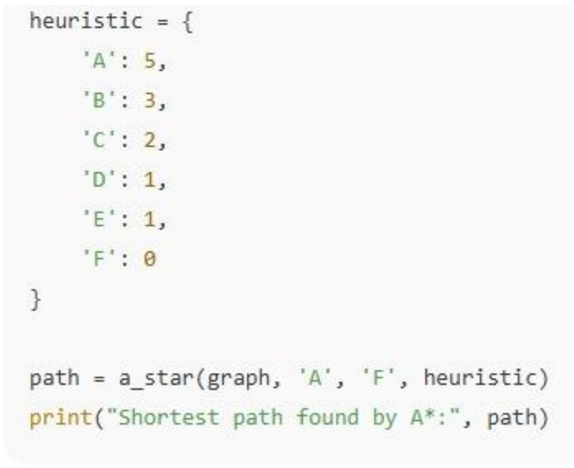
Algorithm:

1. Initialize an open list (priority queue) with the start node.
2. Keep a g-score (cost from start) and f-score (estimated total cost).
3. While the open list is not empty:
   a. Select the node with the lowest f-score.
   b. If it's the goal node, reconstruct and return the path.
   c. For each neighbour:
      - Calculate tentative g-score.

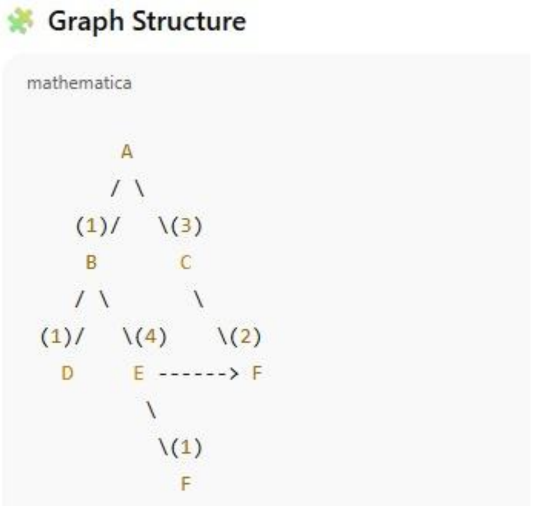
    - If it’s better than the previous score, update and enqueue.
4. Repeat until the goal is reached or all nodes are explored.

CODE:

```
from queue import PriorityQueue

def a_star(graph, start, goal, heuristic):
    open_list = PriorityQueue()
    open_list.put((0, start))
    came_from = {}
    g_score = {node: float('inf') for node in graph}
    g_score[start] = 0

    while not open_list.empty():
        _, current = open_list.get()
        if current == goal:
            break

        for neighbor, cost in graph[current]:
            temp_g = g_score[current] + cost
            if temp_g < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = temp_g
                f_score = temp_g + heuristic[neighbor]
                open_list.put((f_score, neighbor))

    path = [goal]
    while goal in came_from:
        goal = came_from[goal]
        path.append(goal)
    path.reverse()
    return path

graph = {
    'A': [('B', 1), ('C', 3)],
    'B': [('D', 1), ('E', 4)],
    'C': [('F', 2)],
    'D': [],
    'E': [('F', 1)],
    'F': []
}

heuristic = {'A':5, 'B':3, 'C':4, 'D':2, 'E':1, 'F':0}

print("A* Path:", a_star(graph, 'A', 'F', heuristic))
```



```
heuristic = {
    'A': 5,
    'B': 3,
    'C': 2,
    'D': 1,
    'E': 1,
    'F': 0
}

path = a_star(graph, 'A', 'F', heuristic)
print("Shortest path found by A*:", path)
```



Graph Structure

```
mathematica

        A
       / \
   (1)/   \(3)
     B       C
    / \       \
(1)/   \(4)    \(2)
  D     E ------> F
         \
          \(1)
           F
```

OUTPUT:

```
A* Path: ['A', 'B', 'E', 'F']
```

CONCLUSION:

A* combines the benefits of Dijkstra’s algorithm and heuristic search, providing an optimal and efficient pathfinding method for robotics navigation.

EXPERIMENT 6

IMPLEMENT AO ALGORITHM USING PYTHON

AIM:
To implement the AO* (And-Or Star) algorithm in Python for solving problems represented as And-Or graphs.

OBJECTIVES:

1. To understand problem representation using And-Or graphs.
2. To implement AO* for optimal solution finding.
3. To explore its use in problem decomposition and planning.

THEORY:
AO* is an informed search algorithm used for And-Or graphs, where nodes can represent multiple subproblems (AND nodes) or alternative solutions (OR nodes). It uses a heuristic cost function and recursively updates the cost of nodes to find the best solution graph.

Applications:

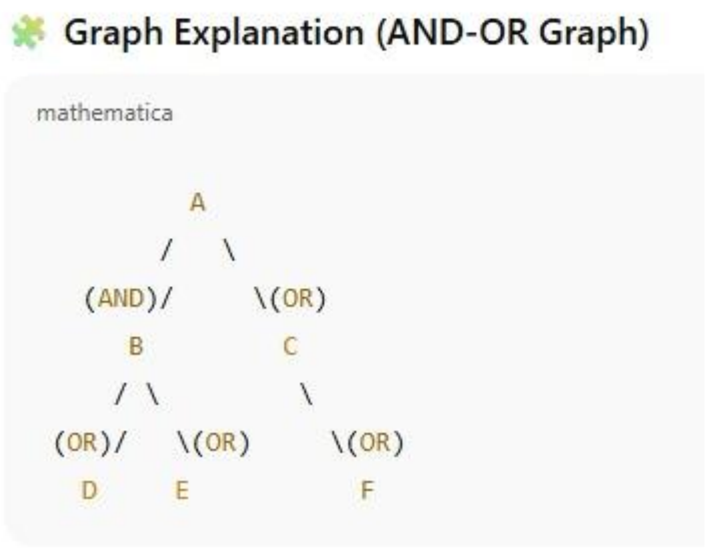
- Planning and reasoning systems.
- Problem-solving in hierarchical task networks (HTN).

Algorithm (AO):*

1. Start from the root node.
2. Expand the most promising node (based on heuristic value).
3. For each child node:
    - If it's an AND node, sum the costs of all children.
    - If it's an OR node, select the minimum cost among children.
4. Update the heuristic value of the parent node.
5. Continue until all paths to terminal nodes are resolved.

CODE:

```
graph = {
    'A': [('B', 'AND', 1), ('C', 'OR', 3)],
    'B': [('D', 'OR', 1), ('E', 'OR', 1)],
    'C': [('F', 'OR', 2)],
    'D': [], 'E': [], 'F': []
}

heuristic = {'A':3, 'B':2, 'C':1, 'D':0, 'E':0, 'F':0}

def ao_star(node):
    if not graph[node]:
        return heuristic[node]
    min_cost = float('inf')
    for child, relation, cost in graph[node]:
        total = cost + heuristic[child]
        if total < min_cost:
            min_cost = total
    heuristic[node] = min_cost
    return heuristic[node]

print("Updated heuristic value of root (A):", ao_star('A'))
```



🧩 Graph Explanation (AND-OR Graph)

mathematica

```
          A
        /   \
  (AND)/     \(OR)
     B         C
    / \         \
(OR)/   \(OR)     \(OR)
  D     E           F
```

OUTPUT:

Updated heuristic value of root (A): 2

CONCLUSION:

The AO* algorithm efficiently handles complex problem decompositions by combining AND/OR reasoning, making it powerful for AI planning tasks.

EXPERIMENT 7

IMPLEMENT TIC-TAC-TOE GAME USING PYTHON

AIM:
To implement a two-player Tic-Tac-Toe game using Python.

OBJECTIVES:

1. To understand game logic and state representation.
2. To develop a simple interactive AI-based game.
3. To practice conditional structures and loops in Python.

THEORY:
Tic-Tac-Toe is a two-player game on a 3×3 grid where players alternate marking cells with "X" or "O". The player who aligns three marks horizontally, vertically, or diagonally wins.

This experiment demonstrates logic design, turn management, and state evaluation.

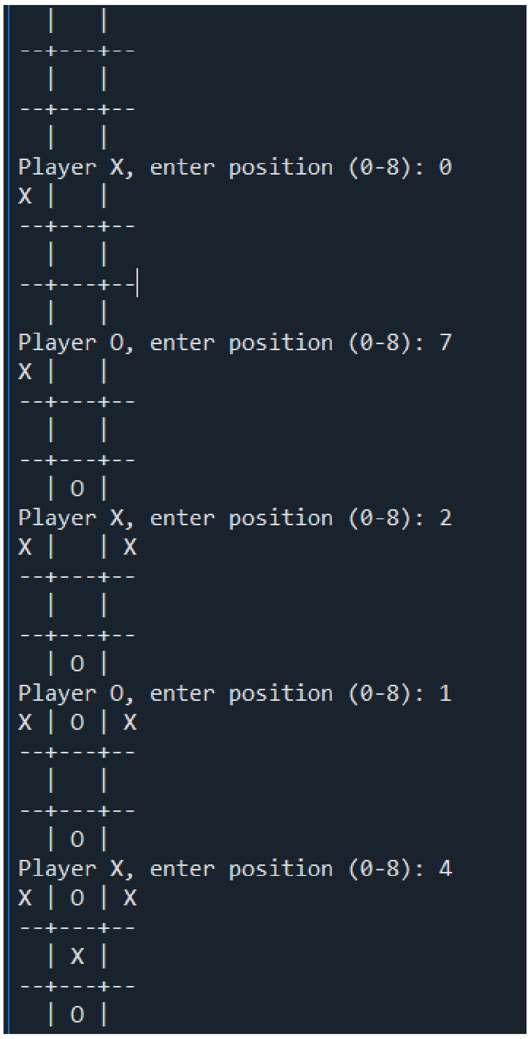
Algorithm (Tic-Tac-Toe):

1. Initialize an empty 3×3 board.
2. Assign symbols to players (X and O).
3. Repeat until all cells are filled or one player wins:
    - Display the board.
    - Take the current player’s move.
    - Update the board.
    - Check for a winner or draw.
    - Switch turns.
4. Display the final result.

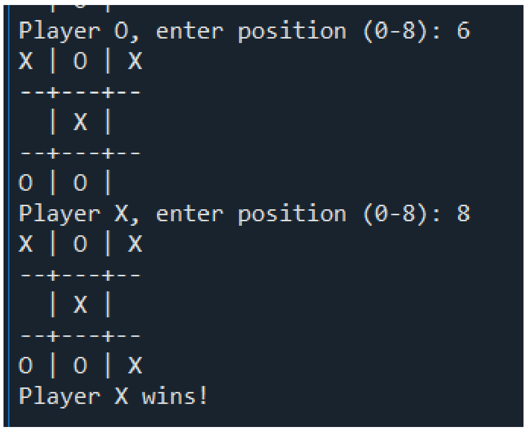
CODE:

```
board = [' ']*9


def print_board():
    for i in range(0, 9, 3):
        print(board[i] + ' | ' + board[i+1] + ' | ' + board[i+2])
        if i < 6: print('--+---+--')


def check_winner(symbol):
    win_conditions = [(0,1,2),(3,4,5),(6,7,8),
                      (0,3,6),(1,4,7),(2,5,8),
                      (0,4,8),(2,4,6)]
    return any(board[a]==board[b]==board[c]==symbol for a,b,c in win_conditions)


def tic_tac_toe():
    turn = 'X'
    for _ in range(9):
        print_board()
        move = int(input(f"Player {turn}, enter position (0-8): "))
        if board[move] == ' ':
            board[move] = turn
            if check_winner(turn):
                print_board()
                print(f"Player {turn} wins!")
                return
            turn = 'O' if turn == 'X' else 'X'
    print_board()
    print("It's a draw!")


tic_tac_toe()
```

OUTPUT:
Displays the interactive board and announces the winner or draw after all moves.



```
  |   |
--+---+--
  |   |
--+---+--
  |   |
Player X, enter position (0-8): 0
X |   |
--+---+--
  |   |
--+---+--
  |   |
Player O, enter position (0-8): 7
X |   |
--+---+--
  |   |
--+---+--
  | O |
Player X, enter position (0-8): 2
X |   | X
--+---+--
  |   |
--+---+--
  | O |
Player O, enter position (0-8): 1
X | O | X
--+---+--
  |   |
--+---+--
  | O |
Player X, enter position (0-8): 4
X | O | X
--+---+--
  | X |
--+---+--
  | O |
```



```
Player O, enter position (0-8): 6
X | O | X
--+---+--
  | X |
--+---+--
O | O |
Player X, enter position (0-8): 8
X | O | X
--+---+--
  | X |
--+---+--
O | O | X
Player X wins!
```

CONCLUSION:
The Tic-Tac-Toe game demonstrates the application of conditional logic and iterative programming for interactive systems.

EXPERIMENT 8

IMPLEMENT TOWER OF HANOI USING PYTHON

AIM:
To write a Python program for solving the Tower of Hanoi problem using recursion.

OBJECTIVES:

1. To understand the recursive problem-solving approach.
2. To visualize the process of moving disks between towers.
3. To demonstrate algorithmic reasoning and recursion.

THEORY:
The Tower of Hanoi puzzle involves moving a stack of disks from one peg to another using a third as an auxiliary, following three rules:

1. Only one disk can be moved at a time.
2. A disk can only be placed on a larger disk.
3. Only the top disk of a tower can be moved.

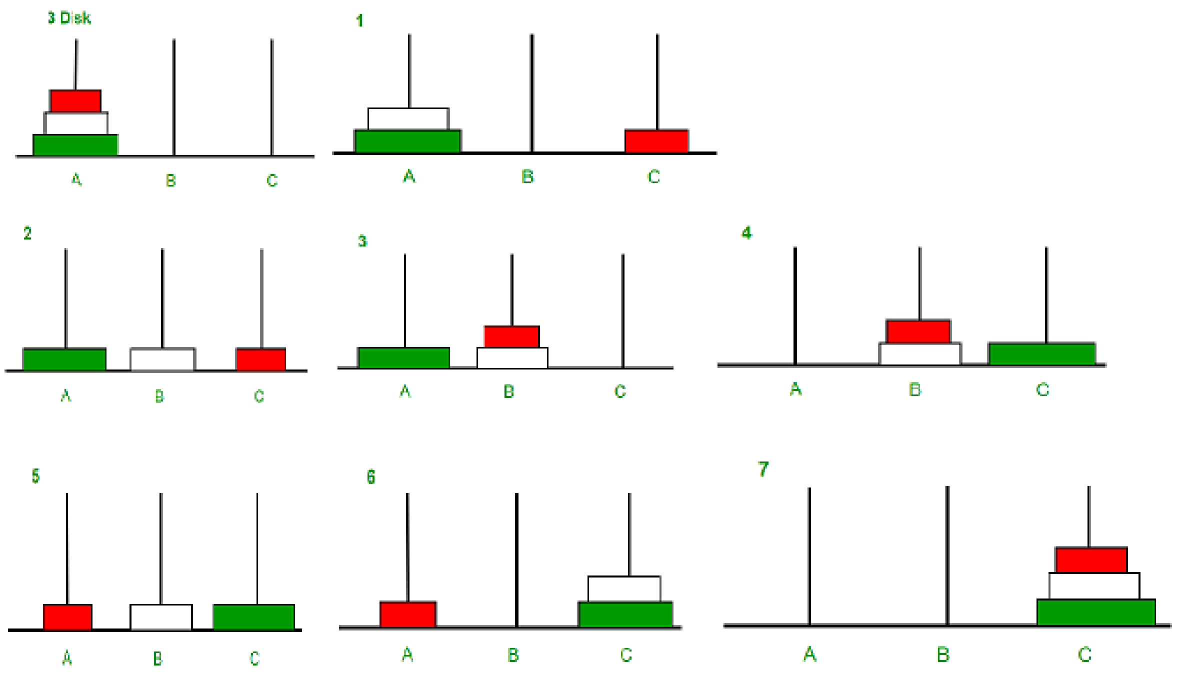
The recursive solution involves breaking down the problem into smaller subproblems.

Algorithm:

1. If there is only one disk, move it directly to the target peg.
2. Recursively move n-1 disks from source to auxiliary.
3. Move the remaining disk to the target peg.
4. Recursively move the n-1 disks from auxiliary to target.

CODE:

```
def tower_of_hanoi(n, source, target, auxiliary):
    if n == 1:
        print(f"Move disk 1 from {source} to {target}")
        return
    tower_of_hanoi(n-1, source, auxiliary, target)
    print(f"Move disk {n} from {source} to {target}")
    tower_of_hanoi(n-1, auxiliary, target, source)

n = 3
tower_of_hanoi(n, 'A', 'C', 'B')
```



OUTPUT:

```
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
```

CONCLUSION:

The Tower of Hanoi illustrates recursive problem-solving, where complex tasks are divided into smaller, repeatable steps.