🧠 1️⃣ Depth First Search (DFS)

AIM:

To implement the **Depth First Search (DFS)** algorithm using recursion in Python.

THEORY:

Depth First Search (DFS) is a **graph traversal algorithm** that explores as far as possible along each branch before backtracking.

- It uses a **stack (implicitly in recursion)**.
- It goes **deep** into one branch before moving to another.
- It is used in **path finding, topological sorting, and cycle detection**.

PROGRAM:

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

def dfs_recursive(graph, node, visited=None):
    if visited is None:
        visited = set()
    if node not in visited:
        print(node, end=' ')
        visited.add(node)
        for neighbour in graph[node]:
            dfs_recursive(graph, neighbour, visited)

print("DFS traversal:")
dfs_recursive(graph, 'A')
```

OUTPUT:

```
DFS traversal:
A B D E F C
```

EXPLANATION:

1. The graph is represented using a **dictionary**.
2. The function `dfs_recursive()`:
    - Takes a node and a visited set.
    - Prints the node if not visited.
    - Recursively calls DFS for all connected neighbors.
3. Starting from `A`, it visits:
    - A → B → D → E → F → (backtrack) → C

🌳 2️⃣ Breadth First Search (BFS)

AIM:

To implement **Breadth First Search (BFS)** algorithm using a queue in Python.

THEORY:

BFS explores all the **neighboring nodes** first before moving to the next level.

- It uses a **queue** (FIFO order).
- It's used in **shortest path finding**, **network broadcasting**, etc.

PROGRAM:

```
from collections import deque

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

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(graph[node])

print("\nBFS traversal:")
bfs(graph, 'A')
```

OUTPUT:

```
BFS traversal:
A B C D E F
```

EXPLANATION:

1. Queue initially contains **A**.
2. Visit A → enqueue B, C.
3. Visit B → enqueue D, E.
4. Visit C → enqueue F.
5. Continue until queue is empty.

🚀 3️⃣ *A Search Algorithm**

AIM:

To implement the *A search algorithm** to find the optimal path using heuristic values.

THEORY:

A* is a **best-first search algorithm** used in pathfinding.

- It uses two costs:
  - **g(n)** = cost from start to current node.
  - **h(n)** = heuristic (estimated cost to goal).
  - **f(n) = g(n) + h(n)**
- Selects the node with the **lowest f(n)**.

PROGRAM:

```
from queue import PriorityQueue

def a_star(graph, heuristics, start, goal):
    pq = PriorityQueue()
    pq.put((0 + heuristics[start], 0, start, [start]))  # (f, g, node, path)

    while not pq.empty():
        f, g, node, path = pq.get()

        if node == goal:
            print("A* Path:", " -> ".join(path))
            print("Total Cost:", g)
            return

        for neighbor, cost in graph[node]:
            new_g = g + cost
            new_f = new_g + heuristics[neighbor]
            pq.put((new_f, new_g, neighbor, path + [neighbor]))

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

heuristics = {'A': 7, 'B': 6, 'C': 2, 'D': 1, 'E': 0}

a_star(graph, heuristics, 'A', 'E')
```

OUTPUT:

```
A* Path: A -> C -> E
Total Cost: 6
```

EXPLANATION:

- It calculates cost for every path using **f = g + h**.
- Chooses the minimum f node.
- Finds the optimal path **A** → **C** → **E**.

💧 4️⃣ Water Jug Problem

AIM:

To solve the **Water Jug problem** using DFS approach.

THEORY:

The Water Jug problem finds a way to measure the desired amount using two jugs of different capacities.

- States are represented as pairs `(x, y)` for jug amounts.
- Transitions include filling, emptying, or pouring water.

PROGRAM:

```
def water_jug(x, y, target):
    visited = set()
    stack = [(0, 0)]
    while stack:
        a, b = stack.pop()
        if (a, b) in visited:
            continue
        print((a, b))
        visited.add((a, b))
        if a == target or b == target:
            return
        stack.extend([
            (x, b), (a, y), (0, b), (a, 0),
            (min(x, a + b), max(0, a + b - x)),
            (max(0, a + b - y), min(y, a + b))
        ])

water_jug(4, 3, 2)
```

OUTPUT:

```
(0, 0)
(0, 3)
(3, 0)
(3, 3)
(4, 2)
```

EXPLANATION:

- Starts with both jugs empty.
- Fills, empties, and pours water until one jug has 2 liters.

🧩 5 Meeting Scheduling (Constraint Satisfaction)

AIM:

To find a common meeting day using Constraint Satisfaction.

THEORY:

Each person has a set of available days.
We find the **intersection** of all sets to find a day suitable for everyone.

PROGRAM:

```
people = {
    "A": ["Mon", "Tue"],
    "B": ["Tue", "Wed"],
    "C": ["Tue", "Thu"],
    "D": ["Tue", "Fri"],
```

```
    "E": ["Tue", "Sat"]
}

common = set(next(iter(people.values())))

for days in people.values():
    common &= set(days)

if common:
    print("Meeting scheduled on:", ", ".join(common), "at 10AM, Room-101")
else:
    print("No common day available for the meeting.")
```

OUTPUT:

```
Meeting scheduled on: Tue at 10AM, Room-101
```

EXPLANATION:

- Starts with A's available days.
- Finds intersection across all others.
- Result → Only Tuesday is common.

🧮 6️⃣ Unification Algorithm (AI Logic)

AIM:

To implement the **Unification algorithm** used in predicate logic.

THEORY:

Unification finds a substitution that makes two logical expressions identical.
Used in **Prolog**, **Theorem proving**, and **Inference engines**.

PROGRAM:

```
def unify(x, y, subs={}):
    if subs is None: return None
    elif x == y: return subs
    elif isinstance(x, str) and x.islower():
        return unify_var(x, y, subs)
    elif isinstance(y, str) and y.islower():
        return unify_var(y, x, subs)
    elif isinstance(x, tuple) and isinstance(y, tuple) and len(x) == len(y):
        for a, b in zip(x, y):
            subs = unify(a, b, subs)
            if subs is None: return None
        return subs
    else: return None

def unify_var(var, x, subs):
    if var in subs: return unify(subs[var], x, subs)
    elif x in subs: return unify(var, subs[x], subs)
    else:
        subs2 = subs.copy(); subs2[var] = x
        return subs2

print("Unification:", unify(('P', 'x'), ('P', 'y')))
```

OUTPUT:

```
Unification: {'x': 'y'}
```

EXPLANATION:

- The function checks variable–constant and variable–variable matches.
- Here, both predicates are same, only variable differs → `'x' = 'y'`.

🧠 7️⃣ Inference Engine

AIM:

To implement a **Forward Chaining Inference Engine**.

THEORY:

Inference derives new facts from existing ones using rules.

- **Forward chaining** starts with known facts and applies rules to infer new facts.

PROGRAM:

```
facts = {"bird(tweety)", "cat(sylvester)", "fish(nemo)"}
rules = {"bird(X) -> canfly(X)"}

def infer(facts, rules):
    derived = set()
    for r in rules:
        lhs, rhs = r.split("->")
        lhs = lhs.strip(); rhs = rhs.strip()
        pred, arg = lhs[:-1].split("(")
        for f in facts:
            if f.startswith(pred):
                derived.add(rhs.replace("X", f[f.index("(")+1:-1]))
    return derived

print("Facts:", facts)
print("Inferred:", infer(facts, rules))
```

OUTPUT:

```
Facts: {'bird(tweety)', 'cat(sylvester)', 'fish(nemo)'}
Inferred: {'canfly(tweety)'}
```

EXPLANATION:

- The rule says: *If bird(X), then canfly(X)*.
- Since `bird(tweety)` is a fact → we infer `canfly(tweety)`.

🧩 8️⃣ 8-Puzzle Problem using A*

AIM:

To solve the **8-puzzle problem** using A* algorithm.

THEORY:

- The puzzle consists of 9 tiles (8 numbered + 1 blank).
- Goal: Move tiles to reach target configuration using minimum moves.
- A* is used with heuristic function (misplaced tiles count).

PROGRAM:

```
import heapq

def h(state, goal):
    return sum(s != g for s, g in zip(state, goal))

def astar(start, goal):
    pq = [(h(start, goal), 0, start, [])]
    visited = set()
    while pq:
        f, g, state, path = heapq.heappop(pq)
        if state == goal:
            print("Solution:", path + [state])
            return
        visited.add(state)
        zero = state.index(0)
        moves = [-1, 1, -3, 3]
        for m in moves:
            new = zero + m
            if 0 <= new < 9:
                s = list(state)
                s[zero], s[new] = s[new], s[zero]
                tup = tuple(s)
                if tup not in visited:
                    heapq.heappush(pq, (g + 1 + h(tup, goal), g + 1, tup, path +
[state]))

start = (1, 2, 3, 4, 0, 5, 6, 7, 8)
goal = (1, 2, 3, 4, 5, 6, 7, 8, 0)
astar(start, goal)
```

OUTPUT:

```
Solution: [(1, 2, 3, 4, 0, 5, 6, 7, 8), (1, 2, 3, 4, 5, 0, 6, 7, 8), (1, 2, 3, 4,
5, 6, 0, 7, 8), (1, 2, 3, 4, 5, 6, 7, 0, 8), (1, 2, 3, 4, 5, 6, 7, 8, 0)]
```

EXPLANATION:

- The blank (0) moves to reach the goal.
- A* chooses optimal path using heuristic (tiles misplaced).