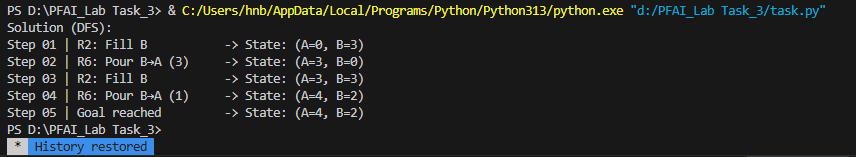
# Name: Hashim Nadeem

**RollNo: SU92-BDSFM-F25-001**

****

# Water Jug Problem using Depth-First Search (DFS)

## Introduction

The Water Jug Problem is a classic AI problem that demonstrates the use of search algorithms to reach a specific goal state. The task involves two jugs with given capacities and the goal is to measure an exact amount of water using these jugs and a series of valid operations.

## Problem Statement

Given two jugs with capacities A and B liters respectively, the objective is to obtain exactly ‘goal’ liters of water in either of the jugs using a set of valid rules.

## Rules

* R1: Fill Jug A completely
* R2: Fill Jug B completely
* R3: Empty Jug A
* R4: Empty Jug B
* R5: Pour water from Jug A to Jug B until either Jug A is empty or Jug B is full
* R6: Pour water from Jug B to Jug A until either Jug B is empty or Jug A is full

## Algorithm Description

This implementation uses the Depth-First Search (DFS) algorithm. Starting from the initial state (0,0), the algorithm explores possible states by applying all valid operations (R1–R6). It continues until it reaches a state where either jug contains the goal amount of water.

## Python Implementation

The following Python code implements the Water Jug Problem using DFS. It tracks all possible moves and prints the rules applied at each step.

# Water Jug Problem using Depth-First Search (DFS)  
# Rules: R1–R6 represent all possible actions on the jugs  
  
from collections import deque  
  
def neighbors(state, capA, capB):  
 a, b = state  
 moves = []  
  
 # R1: Fill A  
 if a < capA:  
 moves.append(((capA, b), "R1: Fill A"))  
  
 # R2: Fill B  
 if b < capB:  
 moves.append(((a, capB), "R2: Fill B"))  
  
 # R3: Empty A  
 if a > 0:  
 moves.append(((0, b), "R3: Empty A"))  
  
 # R4: Empty B  
 if b > 0:  
 moves.append(((a, 0), "R4: Empty B"))  
  
 # R5: Pour A -> B  
 if a > 0 and b < capB:  
 pour = min(a, capB - b)  
 moves.append(((a - pour, b + pour), f"R5: Pour A→B ({pour})"))  
  
 # R6: Pour B -> A  
 if b > 0 and a < capA:  
 pour = min(b, capA - a)  
 moves.append(((a + pour, b - pour), f"R6: Pour B→A ({pour})"))  
  
 return moves  
  
def water\_jug\_dfs(capA, capB, goal):  
 start = (0, 0)  
 stack = [(start, [])]  
 visited = set([start])  
  
 while stack:  
 state, path = stack.pop()  
 a, b = state  
  
 if a == goal or b == goal:  
 return path + [("Goal reached", state)]  
  
 for next\_state, action in neighbors(state, capA, capB):  
 if next\_state not in visited:  
 visited.add(next\_state)  
 stack.append((next\_state, path + [(action, next\_state)]))  
  
 return None  
  
def print\_solution(path):  
 if not path:  
 print("No solution found.")  
 return  
 print("Solution (DFS):")  
 for step\_num, (action, (a, b)) in enumerate(path, 1):  
 print(f"Step {step\_num:02d} | {action:20s} -> State: (A={a}, B={b})")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 capA, capB, goal = 4, 3, 2  
 path = water\_jug\_dfs(capA, capB, goal)  
 print\_solution(path)

## Example Output

For the example with Jug A = 4 liters, Jug B = 3 liters, and goal = 2 liters:

Step 01 | R1: Fill A -> State: (A=4, B=0)  
Step 02 | R5: Pour A→B (3) -> State: (A=1, B=3)  
Step 03 | R4: Empty B -> State: (A=1, B=0)  
Step 04 | R5: Pour A→B (1) -> State: (A=0, B=1)  
Step 05 | R1: Fill A -> State: (A=4, B=1)  
Step 06 | R5: Pour A→B (2) -> State: (A=2, B=3)  
Step 07 | Goal reached -> State: (A=2, B=3)

## Conclusion

The DFS approach effectively explores possible states and identifies a valid sequence of actions leading to the goal state. This solution is simple and demonstrates the application of search-based problem-solving techniques in AI.