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

The script solves the classic **Water Jug Problem** using a **Depth-First Search (DFS)** approach. The problem involves two jugs with given capacities and the goal of measuring an exact amount of water.

**Code Breakdown**

**1. Function Definition:**

def water\_jug\_dfs(CAP\_A, CAP\_B, TARGET):

* CAP\_A: Capacity of the first jug.
* CAP\_B: Capacity of the second jug.
* TARGET: The exact amount of water to measure.

**2. Initializing Variables**

start\_state = (0, 0) # Both jugs start empty

path = [] # Stores the sequence of steps taken

visited = set() # Keeps track of visited states

* start\_state represents the initial state (0, 0), where both jugs are empty.
* path stores the sequence of steps leading to the solution.
* visited is used to avoid revisiting already explored states.

**3. Depth-First Search (DFS)**

def dfs(current\_state):

* The function explores all possible next states from the current state recursively.

**Handling the Goal Condition**

if a == TARGET or b == TARGET:

* If either jug contains the target amount, the solution is found, and the sequence of steps is printed.

**Generating Next Possible States**

next\_states = []

next\_states.append((CAP\_A, b)) # Fill jug A completely

next\_states.append((a, CAP\_B)) # Fill jug B completely

next\_states.append((0, b)) # Empty jug A

next\_states.append((a, 0)) # Empty jug B

* The algorithm generates new states by:
  + Filling either jug to its full capacity.
  + Emptying either jug.

**Pouring Water Between Jugs**

pour\_amount = min(a, CAP\_B - b) # Maximum transferable from A to B

new\_a = a - pour\_amount

new\_b = b + pour\_amount

next\_states.append((new\_a, new\_b))

* Transfers water from **jug A to jug B** until:
  + Jug B is full.
  + Jug A runs out of water.

pour\_amount = min(b, CAP\_A - a) # Maximum transferable from B to A

new\_b = b - pour\_amount

new\_a = a + pour\_amount

next\_states.append((new\_a, new\_b))

* Transfers water from **jug B to jug A** in a similar manner.

**Recursive DFS Call**

for state in next\_states:

if state not in visited:

if dfs(state):

return True

* The function recursively calls itself to explore all possible states.
* If a state leads to a solution, the function returns True to stop further exploration.

**Backtracking**

path.pop()

* If a solution is not found in the current path, the last state is removed, and DFS backtracks to explore other possibilities.

**4. Handling No Solution Case**

if not found\_solution:

print(f"No solution found for measuring exactly {TARGET} liters with jugs of capacity {CAP\_A} and {CAP\_B}.")

* If DFS completes without finding a solution, it prints a failure message.

**Execution Example**

The script runs with:

water\_jug\_dfs(4, 3, 2)

* **Jug A**: 4 liters
* **Jug B**: 3 liters
* **Target**: 2 liters

**Expected Output (One Possible Solution)**

Solution found!

Sequence of states leading to the solution:

(0, 0)

(4, 0)

(1, 3)

(1, 0)

(0, 1)

(4, 1)

(2, 3)

(2, 0)

* This sequence shows how the two jugs are filled, emptied, and poured into each other to reach the target.

**Summary**

* The program efficiently finds a solution (if one exists) using **DFS**.
* It keeps track of visited states to avoid loops.
* If a solution is found, it prints the sequence of moves; otherwise, it reports that no solution exists.

**Output:**

