USING BFS

```
from collections import deque

def Solution(a, b, target):
    m = {}
    isSolvable = False
    path = []
    q = deque()
    q.append((0, 0))

    while len(q) > 0:
        u = q.popleft()

        if (u[0], u[1]) in m:
            continue

        if u[0] > a or u[1] > b or u[0] < 0 or u[1] < 0:
            continue

        path.append([u[0], u[1]])
        m[(u[0], u[1])] = 1

        if u[0] == target or u[1] == target:
            isSolvable = True

            if u[0] == target:
                if u[1] != 0:
                    path.append([u[0], 0])
            else:
                if u[0] != 0:
                    path.append([0, u[1]])

            for state in path:
                print("(", state[0], ",", state[1], ")")
            break

        q.append((u[0], b))
        q.append((a, u[1]))

        for ap in range(max(a, b) + 1):

            c = u[0] + ap
            d = u[1] - ap
            if c == a or (d == 0 and d >= 0):
                q.append((c, d))
```

```
c = u[0] - ap
          d = u[1] + ap
          if (c == 0 and c >= 0) or d == b:
             q.append((c, d))

       q.append((a, 0))
       q.append((0, b))

    if not isSolvable:
       print("Solution not possible")


if __name__ == "__main__":
    Jug1 = int(input("Enter the capacity of Jug1: "))
    Jug2 = int(input("Enter the capacity of Jug2: "))
    target = int(input("Enter the target: "))
    print("Path from initial state to solution state ::")
    Solution(Jug1, Jug2, target)
```

OUTPUT:

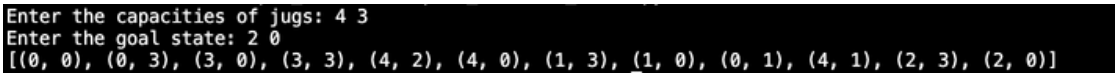
```
Enter the capacity of Jug1: 4
Enter the capacity of Jug2: 3
Enter the target: 2
Path from initial state to solution state ::
( 0 , 0 )
( 0 , 3 )
( 4 , 0 )
( 4 , 3 )
( 3 , 0 )
( 1 , 3 )
( 3 , 3 )
( 4 , 2 )
( 0 , 2 )
```

USING DFS:

```
class Node:
    def __init__(self, state, parent):
        self.state = state
        self.parent = parent

    def get_child_nodes(self, capacities):
        a, b = self.state
        max_a, max_b = capacities
        children = []

        children.append(Node((max_a, b), self))
        children.append(Node((a, max_b), self))
        children.append(Node((0, b), self))
        children.append(Node((a, 0), self))

        if a + b >= max_b:
            children.append(Node((a - (max_b - b), max_b), self))
        else:
            children.append(Node((0, a + b), self))

        if a + b >= max_a:
            children.append(Node((max_a, b - (max_a - a)), self))
        else:
            children.append(Node((a + b, 0), self))

        return children


def dfs(start_state, goal_state, capacities):
    start_node = Node(start_state, None)
    visited = set()
    stack = [start_node]

    while stack:
        node = stack.pop()

        if node.state == goal_state:
            path = []
            while node.parent:
                path.append(node.state)
                node = node.parent
            path.append(start_state)
            path.reverse()
            return path
```

```
        if node.state not in visited:
            visited.add(node.state)
            for child in node.get_child_nodes(capacities):
            stack.append(child)

    return None


start_state = (0, 0)

a, b = map(int, input("Enter the capacities of jugs: ").split())
c, d = map(int, input("Enter the goal state: ").split())

goal_state = (c, d)
capacities = (a, b)

path = dfs(start_state, goal_state, capacities)
print(path)
```

OUTPUT:



MEMOIZATON:

```
from collections import defaultdict

jug1, jug2, aim = 4, 3, 2
visited = defaultdict(lambda: False)

def waterJugSolver(amt1, amt2):

    if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):
        print(amt1, amt2)
        return True

    if visited[(amt1, amt2)] == False:
        print(amt1, amt2)
        visited[(amt1, amt2)] = True

        return (
            waterJugSolver(0, amt2) or
            waterJugSolver(amt1, 0) or
            waterJugSolver(jug1, amt2) or
            waterJugSolver(amt1, jug2) or
            waterJugSolver(
                amt1 + min(amt2, jug1 - amt1),
                amt2 - min(amt2, jug1 - amt1)
            ) or
            waterJugSolver(
                amt1 - min(amt1, jug2 - amt2),
                amt2 + min(amt1, jug2 - amt2)
            )
        )
    else:
        return False


print("Steps:")
waterJugSolver(0, 0)
```

OUTPUT: