Exercise 1:

```
from collections import deque

def BFS(a, b, target):

    # Map is used to store the states, every
    # state is hashed to binary value to
    # indicate either that state is visited
    # before or not
    m = {}
    isSolvable = False
    path = []

    # Queue to maintain states
    q = deque()

    # Initialing with initial state
    q.append((0, 0))

    while (len(q) > 0):

        # Current state
        u = q.popleft()

        #q.pop() #pop off used state

        # If this state is already visited
        if ((u[0], u[1]) in m):
            continue
```

```
# Doesn't met jug constraints
if ((u[0] > a or u[1] > b or
	u[0] < 0 or u[1] < 0)):
	continue

# Filling the vector for constructing
# the solution path
path.append([u[0], u[1]])

# Marking current state as visited
m[(u[0], u[1])] = 1

# If we reach solution state, put ans=1
if (u[0] == target or u[1] == target):
	isSolvable = True

	if (u[0] == target):
		if (u[1] != 0):

			# Fill final state
			path.append([u[0], 0])
	else:
		if (u[0] != 0):

			# Fill final state
			path.append([0, u[1]])

	# Print the solution path
```

```
        sz = len(path)
        for i in range(sz):
            print("(", path[i][0], ",",
                    path[i][1], ")")
        break

    # If we have not reached final state
    # then, start developing intermediate
    # states to reach solution state
    q.append([u[0], b]) # Fill Jug2
    q.append([a, u[1]]) # Fill Jug1

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

        # Pour amount ap from Jug2 to Jug1
        c = u[0] + ap
        d = u[1] - ap

        # Check if this state is possible or not
        if (c == a or (d == 0 and d >= 0)):
            q.append([c, d])

        # Pour amount ap from Jug 1 to Jug2
        c = u[0] - ap
        d = u[1] + ap

        # Check if this state is possible or not
        if ((c == 0 and c >= 0) or d == b):
            q.append([c, d])
```

```
                # Empty Jug2
                q.append([a, 0])

                # Empty Jug1
                q.append([0, b])

        # No, solution exists if ans=0
        if (not isSolvable):
                print ("No solution")

# Driver code
if __name__ == '__main__':

        Jug1, Jug2, target = 4, 3, 2
        print("Path from initial state "
                "to solution state ::")

        BFS(Jug1, Jug2, target)
```

Output:

```
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 )
```

Exercise 2:

```
# This function is used to initialize the
# dictionary elements with a default value.
from collections import defaultdict

# jug1 and jug2 contain the value
# for max capacity in respective jugs
# and aim is the amount of water to be measured.
jug1, jug2, aim = 4, 3, 2

# Initialize dictionary with
# default value as false.
visited = defaultdict(lambda: False)

# Recursive function which prints the
# intermediate steps to reach the final
# solution and return boolean value
# (True if solution is possible, otherwise False).
# amt1 and amt2 are the amount of water present
# in both jugs at a certain point of time.
def waterJugSolver(amt1, amt2):

    # Checks for our goal and
    # returns true if achieved.
    if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):
        print(amt1, amt2)
        return True

    # Checks if we have already visited the
    # combination or not. If not, then it proceeds further.
```

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

        # Changes the boolean value of
        # the combination as it is visited.
        visited[(amt1, amt2)] = True

        # Check for all the 6 possibilities and
        # see if a solution is found in any one of them.
        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))))

    # Return False if the combination is
    # already visited to avoid repetition otherwise
    # recursion will enter an infinite loop.
    else:
        return False

print("Steps: ")

# Call the function and pass the
# initial amount of water present in both jugs.
waterJugSolver(0, 0)
```

output:

```
Steps:
0 0
4 0
4 3
0 3
3 0
3 3
4 2
0 2

True
```

Exercise 3:

```
# Python3 implementation of program to count
# minimum number of steps required to measure
# d litre water using jugs of m liters and n
# liters capacity.
def gcd(a, b):
	if b==0:
		return a
	return gcd(b, a%b)


''' fromCap -- Capacity of jug from which
				water is poured
toCap -- Capacity of jug to which
				water is poured
d		-- Amount to be measured '''
def Pour(toJugCap, fromJugCap, d):
```

```
# Initialize current amount of water
# in source and destination jugs
fromJug = fromJugCap
toJug = 0

# Initialize steps required
step = 1
while ((fromJug is not d) and (toJug is not d)):


    # Find the maximum amount that can be
    # poured
    temp = min(fromJug, toJugCap-toJug)

    # Pour 'temp' liter from 'fromJug' to 'toJug'
    toJug = toJug + temp
    fromJug = fromJug - temp

    step = step + 1
    if ((fromJug == d) or (toJug == d)):
        break

    # If first jug becomes empty, fill it
    if fromJug == 0:
        fromJug = fromJugCap
        step = step + 1

    # If second jug becomes full, empty it
    if toJug == toJugCap:
```

```
                    toJug = 0
                    step = step + 1

        return step

# Returns count of minimum steps needed to
# measure d liter
def minSteps(n, m, d):
        if m> n:
                temp = m
                m = n
                n = temp

        if (d%(gcd(n,m)) != 0):
                return -1

        # Return minimum two cases:
        # a) Water of n liter jug is poured into
        # m liter jug
        return(min(Pour(n,m,d), Pour(m,n,d)))

# Driver code
if __name__ == '__main__':

        n = 3
        m = 5
        d = 4

        print('Minimum number of steps required is',
```

minSteps(n, m, d))

output:

```
Minimum number of steps required is 6
```