365. Water and Jug Problem

Medium | Depth-First Search | Breadth-First Search | Math | Leetcode Link

Problem Description

In the given problem, you have two jugs with specific capacities - one can hold `jug1Capacity` liters and the other can hold `jug2Capacity` liters. With an unlimited water supply, your task is to figure out a way to measure exactly `targetCapacity` liters of water by either filling, emptying, or transferring water between the two jugs. The goal is to have the combined water in both jugs equal the `targetCapacity`. The operations you can perform are:

1. Filling up either jug to its full capacity with water.
2. Emptying any of the jugs completely.
3. Pouring water from one jug into the other until the jug from which you are pouring is empty, or the jug being filled is full.

You are asked to determine if it is possible to achieve the `targetCapacity` through these operations.

Intuition

The solution uses a property from number theory related to the Greatest Common Divisor (GCD). The core intuition hinges on the idea that a certain amount of water can only be measured if it is a multiple of the GCD of the two jug capacities.

This is based on the mathematical theorem which states that for any two integers, a and b, and their GCD g, any integer of the form `ax + by` is a multiple of g. Here, `x` and `y` can be any integer, including negative integers, zero, or positive integers.

In terms of the jugs problem:

- `a` and `b` represent the capacities of the two jugs.
- `g` represents the GCD of the two capacities.
- `x` and `y` represent the number of times we fill or empty a jug.

Therefore, we can measure `targetCapacity` if and only if it's a multiple of the GCD of the two jug capacities.

We should check that the total capacity of the two jugs is at least the `targetCapacity`, because if it isn't, there's no way to measure out that amount of water. Additionally, if either jug has a capacity of 0, the only achievable measurements are either 0 or the combined capacity of both jugs.

In the provided solution, we check that the sum of both jugs' capacities is greater than or equal to the `targetCapacity`. If not, we return `False` because the target cannot be measured. If one jug is of 0 capacity, we then check if it's possible to measure the target - it can only be measured if the target is 0 or equal to the sum of both capacities. Lastly, we use the `%` operator to check if `targetCapacity` is divisible by the GCD of the two jug capacities, which is calculated using the built-in `gcd` function.

Solution Approach

The solution provided is quite straightforward and primarily relies on mathematical reasoning. Here's a step-by-step breakdown of the implementation strategy using the algorithms, data structures, or patterns referenced in the code:

1. **Initial Checks**: First, the implementation checks if the combined capacity of both jugs is less than the `targetCapacity`. If this is true, it immediately returns `False`, as it is impossible to measure more water than the total volume that both jugs can hold.

2. **Handle Zero Capacity Case**: The code then checks if either of the jugs has a capacity of 0. If one jug has zero capacity, then there are only two possibilities to achieve the `targetCapacity`: either it is exactly 0 (which is always achievable), or it equals the non-zero capacity of the other jug. This is done by checking if `targetCapacity` is 0 or if `targetCapacity` equals `jug1Capacity + jug2Capacity`.

3. **Mathematical Theorem Application**: The main part of the solution relies on the property of the greatest common divisor (GCD). According to the theorem, a linear combination of two numbers `a` and `b` (`ax + by` for some integers x and y) can make any multiple of the GCD of `a` and `b`. In the context of our jugs, this means that we can only measure quantities that are multiples of the GCD of the two jug capacities.

4. **Using Python's gcd Function**: The code uses Python's built-in `gcd` function from the `math` module to find the GCD of the capacities of the jugs.

5. **Final Check with Modulus Operator**: With the GCD calculated, the final step is to check if `targetCapacity` is a multiple of the GCD. This is achieved by checking if the remainder of the division of `targetCapacity` by the GCD (`targetCapacity % gcd(jug1Capacity, jug2Capacity)`) is 0. If the remainder is 0, then `targetCapacity` is a multiple of the GCD, and the function returns `True`. Otherwise, if there is a non-zero remainder, it implies `targetCapacity` cannot be formed by any combination of the two jug capacities according to the theorem mentioned, and the function returns `False`.

No advanced data structures or patterns are necessary for this approach, as the problem can be solved solely through arithmetic and logical operations. The elegance of the solution lies in its use of a well-known mathematical principle to reduce what might seem like a complex, operation-based problem to a simple modulo operation.

Here is the essential part of the code encapsulating the solution logic:

```python
class Solution:
    def canMeasureWater(
        self, jug1Capacity: int, jug2Capacity: int, targetCapacity: int
    ) -> bool:
        if jug1Capacity + jug2Capacity < targetCapacity:
            return False
        if jug1Capacity == 0 or jug2Capacity == 0:
            return targetCapacity == 0 or jug1Capacity + jug2Capacity == targetCapacity
        return targetCapacity % gcd(jug1Capacity, jug2Capacity) == 0
```

Each check corresponds to a logical step in our step-by-step approach, ensuring that the problem is tackled efficiently and correctly.

Example Walkthrough

Let's go through an example to illustrate the solution approach with actual numbers. Suppose we have `jug1Capacity = 3` liters, `jug2Capacity = 5` liters, and our `targetCapacity = 4` liters. We want to find out if we can measure exactly 4 liters using these two jugs.

Here are the steps we would take according to our solution approach:

1. **Initial Checks**: We first check if `jug1Capacity + jug2Capacity` is less than `targetCapacity`. In our case, `3 + 5` is not less than `4`, so we can proceed.

2. **Handle Zero Capacity Case**: Next, we check for zero capacities. Neither jug has zero capacity (`jug1Capacity = 3`, `jug2Capacity = 5`), so this condition does not apply to our example.

3. **Mathematical Theorem Application**: We know that we can only measure `targetCapacity` if it is a multiple of the GCD of `jug1Capacity` and `jug2Capacity`. So we need to find the GCD of 3 and 5.

4. **Using Python's gcd Function**: We use the Python `gcd` function to find the GCD of 3 and 5, which is `1` because 3 and 5 are co-prime numbers.

5. **Final Check with Modulus Operator**: We perform the final check: `targetCapacity % gcd(jug1Capacity, jug2Capacity)`. In our example, we check `4 % 1`. Since 4 is a multiple of 1, `4 % 1` equals `0`. This means the remainder of dividing 4 by the GCD, which is 1, equals 0.

Since the final modulus check passes, the function would return `True`, indicating that it is feasible to measure exactly 4 liters using a 3-liter jug and a 5-liter jug. This solution aligns with the well-known "water jug" problem-solving approach, where the operation of transferring water between jugs leads to measuring different volumes which are multiples of the GCD of the jugs' capacities.

Python Solution

```python
from math import gcd  # Importing the gcd function from the math module

class Solution:
    def can_measure_water(self, jug1_capacity: int, jug2_capacity: int, target_capacity: int) -> bool:
        """
        Determine if it is possible to measure exactly the 'target_capacity' amount of water
        using two jugs with capacities 'jug1_capacity' and 'jug2_capacity'.

        Parameters:
            jug1_capacity (int): Capacity of the first jug
            jug2_capacity (int): Capacity of the second jug
            target_capacity (int): The target amount of water we want to measure

        Returns:
            bool: True if the target amount of water can be measured, False otherwise.
        """

        # If the sum of both jugs' capacities is less than the target capacity,
        # it's not possible to measure the target capacity.
        if jug1_capacity + jug2_capacity < target_capacity:
            return False

        # If one jug's capacity is 0, we can only measure the target if it's 0
        # or equal to the capacity of the other jug.
        if jug1_capacity == 0 or jug2_capacity == 0:
            return target_capacity == 0 or jug1_capacity + jug2_capacity == target_capacity

        # If none of the above cases, check if the target capacity is a multiple of
        # the greatest common divisor (GCD) of the two jugs' capacities.
        # This is because we can only measure amounts that are multiples of the GCD.
        return target_capacity % gcd(jug1_capacity, jug2_capacity) == 0

```

Java Solution

```java
class Solution {
    /**
     * Determines if it's possible to measure exactly the target capacity using the two jugs.
     *
     * @param jug1Capacity the capacity of jug 1
     * @param jug2Capacity the capacity of jug 2
     * @param targetCapacity the target capacity to measure
     * @return true if it's possible to measure exact target capacity, false otherwise
     */
    public boolean canMeasureWater(int jug1Capacity, int jug2Capacity, int targetCapacity) {
        // If the sum of both jug capacities is less than the target, it's not possible to measure.
        if (jug1Capacity + jug2Capacity < targetCapacity) {
            return false;
        }
        // If one jug is of 0 capacity, we can only measure the target if
        // it's 0 or equal to the capacity of the non-zero jug.
        if (jug1Capacity == 0 || jug2Capacity == 0) {
            return targetCapacity == 0 || jug1Capacity + jug2Capacity == targetCapacity;
        }
        // The target capacity must be a multiple of the greatest common divisor of the jug capacities.
        return targetCapacity % greatestCommonDivisor(jug1Capacity, jug2Capacity) == 0;
    }

    /**
     * Computes the greatest common divisor (GCD) of two numbers using Euclidean algorithm.
     *
     * @param a the first number
     * @param b the second number
     * @return the greatest common divisor (GCD) of a and b
     */
    private int greatestCommonDivisor(int a, int b) {
        // If the second number is 0, return the first number, otherwise, recursively
        // find the GCD of the second number and the remainder of the first number divided by the second number.
        return b == 0 ? a : greatestCommonDivisor(b, a % b);
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    /* Determines if it is possible to measure exactly 'targetCapacity' liters by using
       two jugs with capacities 'jug1Capacity' and 'jug2Capacity'. */
    bool canMeasureWater(int jug1Capacity, int jug2Capacity, int targetCapacity) {
        // If the sum of both jugs' capacities is less than the target capacity,
        // it's not possible to reach the target capacity.
        if (jug1Capacity + jug2Capacity < targetCapacity) {
            return false;
        }

        // If one jug has 0 capacity, check if the target can be achieved with the other jug alone.
        if (jug1Capacity == 0 || jug2Capacity == 0) {
            return targetCapacity == 0 || jug1Capacity + jug2Capacity == targetCapacity;
        }

        // The target capacity must be a multiple of the greatest common divisor (GCD)
        // of the two jugs' capacities according to the Bezout's identity theorem.
        return targetCapacity % gcd(jug1Capacity, jug2Capacity) == 0;
    }

private:
    // Helper function to calculate the greatest common divisor (GCD) of two numbers 'a' and 'b'.
    int gcd(int a, int b) {
        // If 'b' is zero, 'a' is the GCD. Otherwise, recursively call gcd with 'b' and 'a modulo b'.
        return b == 0 ? a : gcd(b, a % b);
    }
};

```

Typescript Solution

```typescript
// Determines if it is possible to measure exactly 'targetCapacity' liters by using
// two jugs with capacities 'jug1Capacity' and 'jug2Capacity'.
function canMeasureWater(jug1Capacity: number, jug2Capacity: number, targetCapacity: number): boolean {
    // If the sum of both jugs' capacities is less than the target capacity,
    // it's not possible to reach the target capacity.
    if (jug1Capacity + jug2Capacity < targetCapacity) {
        return false;
    }

    // If one jug has 0 capacity, check if the target can be achieved with the other jug alone.
    if (jug1Capacity === 0 || jug2Capacity === 0) {
        return targetCapacity === 0 || jug1Capacity + jug2Capacity === targetCapacity;
    }

    // The target capacity must be a multiple of the greatest common divisor (GCD)
    // of the two jugs' capacities according to the Bezout's identity theorem.
    return targetCapacity % gcd(jug1Capacity, jug2Capacity) === 0;
}

// Helper function to calculate the greatest common divisor (GCD) of two numbers 'a' and 'b'.
function gcd(a: number, b: number): number {
    // If 'b' is zero, 'a' is the GCD. Otherwise, recursively call gcd with 'b' and 'a modulo b'.
    return b === 0 ? a : gcd(b, a % b);
}

```

Time and Space Complexity

The given Python function `canMeasureWater` determines whether it is possible to measure exactly `targetCapacity` liters by using two jugs of capacities `jug1Capacity` and `jug2Capacity`. It does so using a theorem related to the Diophantine equation which states that a target capacity `x` can be measured using two jugs with capacities `m` and `n` if and only if `x` is a multiple of the greatest common divisor (GCD) of `m` and `n`.

Time Complexity:

The time complexity of the function is predominantly determined by the computation of the GCD of `jug1Capacity` and `jug2Capacity`. Here's how the complexity breaks down:

1. The function checks if the sum of the capacities of the two jugs is less than the `targetCapacity`. This comparison is constant time, `O(1)`.
2. Then, it checks if either jug has a `0` capacity, and in such cases, it also performs constant-time comparisons: `O(1)`.
3. Finally, it calculates the GCD of the two jug capacities. The GCD is calculated using Euclid's algorithm, which has a worst-case time complexity of `O(log(min(a, b)))`, where `a` and `b` are `jug1Capacity` and `jug2Capacity`. Since the GCD function is bounded by the smaller of the two numbers, the time complexity for this step is `O(log(min(jug1Capacity, jug2Capacity)))`.

Therefore, the overall time complexity of the function is `O(log(min(jug1Capacity, jug2Capacity)))`.

Space Complexity:

The space complexity of the function is determined by the space used to hold any variables and the stack space used by the recursion (if the implementation of GCD is recursive):

1. Only a fixed number of integer variables are used, and there's no use of any data structures that scale with the input size. This contributes a constant space complexity: `O(1)`.
2. Assuming `gcd` function from the math library is used, which is typically implemented iteratively, the space complexity remains constant as there are no recursive calls stacking up.

Therefore, the overall space complexity of the function is `O(1)` constant space.