780. Reaching Points

Hard Math Leetcode Link

Problem Description

The problem presents an operation that can transform a point `(x, y)` in a two-dimensional grid to either `(x, x + y)` or `(x + y, y)`. Given four integers `sx`, `sy`, `tx`, and `ty`, which represent the starting point `(sx, sy)` and the target point `(tx, ty)`, the task is to determine if it is possible to reach the target point from the starting point by repeatedly applying the transformation.

The only allowed operations are:

1. Add the current `y` value to the `x` value, changing `(x, y)` to `(x + y, y)`.
2. Add the current `x` value to the `y` value, changing `(x, y)` to `(x, x + y)`.

The key challenge is to figure out if, by applying these operations starting from `(sx, sy)`, we can eventually obtain the point `(tx, ty)`. If it's possible, return `true`; otherwise, return `false`.

Intuition

To solve this problem, we can use the reverse thinking approach. Instead of starting from `(sx, sy)` and trying to reach `(tx, ty)` using the allowed operations, we start from `(tx, ty)` and try to work our way backwards to `(sx, sy)`. This approach is valid because the operations are reversible.

We repeatedly subtract the smaller of `tx` and `ty` from the larger until we either pass `sx` and `sy` or land exactly on them. During this process, if `tx` is greater than `ty`, we know that `tx` had to be formed by adding `ty` to a smaller number in the previous step, so we perform `tx %= ty`. The same reasoning applies if `ty` is greater than `tx`.

While doing this, there are three cases:

1. If both `tx` and `ty` are greater than `sx` and `sy` respectively, we continue reducing `tx` and `ty`.
2. If `tx` becomes equal to `sx`, we check if `ty` can be reduced to `sy` by a multiple of `tx`. If so, it means we can reach `(sx, sy)` from `(tx, ty)`.
3. The same check applies if `ty` becomes equal to `sy`.

If none of the conditions are met, then it's not possible to transform `(sx, sy)` to `(tx, ty)` using the operations, so we return `false`.

The time complexity of this approach is O(log(max(tx, ty))), which occurs due to the modulo operation that reduces `tx` and `ty` exponentially.

Solution Approach

The implementation of the solution follows a reverse approach working backward from the target point `(tx, ty)` to the start point `(sx, sy)`. Here's how the solution translates into code:

- The solution starts by initializing a while loop that continues as long as both `tx` and `ty` are larger than `sx` and `sy`, but `tx` is not equal to `ty`. This condition allows us to reduce `tx` and `ty` until either we can't reduce them anymore without going below `sx` or `sy`, or until one of the coordinates matches `sx` or `sy`.
- Inside the loop, we use the `%` (modulo) operator to reverse the operation as follows:
  - If `tx > ty`, then `tx` must have been formed by adding some multiple of `ty` to the previous `x` value, so we perform `tx %= ty`. This reverses the last operation that affected `tx`.
  - If `ty > tx`, then we apply `ty %= tx` for the same reason but in the vertical direction.
- After exiting the loop, either `tx == sx` and/or `ty == sy`, or both `tx` and `ty` have been reduced below `sx` and `sy`. We need to handle both cases to complete the algorithm:
  - If `tx == sx` and `ty == sy` at the same time, we've found the exact reverse path to `(sx, sy)`, and the function returns `true`.
  - If `tx == sx`, then `ty` must be reducible to `sy` by repeatedly subtracting `tx` (the same operation reversed). We check this by ensuring `ty > sy` and `(ty - sy) % tx == 0`. If this is the case, we return `true`.
  - Similarly, if `ty == sy`, we check if `tx > sx` and `(tx - sx) % ty == 0` to determine if `tx` can be reduced to `sx` and return `true` accordingly.
- If none of the conditions are met, it means that we can't reach `(sx, sy)` from `(tx, ty)` by reversing the operations, so the function returns `false`.

This approach is efficient because it avoids the exponential time complexity of trying all possible paths from `(sx, sy)` to `(tx, ty)`. Instead, by taking advantage of the mathematical properties of the operation, it quickly navigates to the solution or determines impossibility with a time complexity of `O(log(max(tx, ty)))`.

Example Walkthrough

Let's consider the problem with a small example:

Imagine our starting point is `(sx, sy) = (2, 4)` and our target point is `(tx, ty) = (9, 4)`. Can we transform the starting point to the target point using the allowed operations?

We apply the reverse approach:

1. We start from `(tx, ty)` which is `(9, 4)` and compare it with `(sx, sy)`. Since `tx > ty`, we perform `tx %= ty`.
2. After applying the modulo operation, `tx` is now `9 % 4 = 1`.
3. Now, our target point `(tx, ty)` has changed to `(1, 4)`.
4. We compare `tx` with `sx`. They are not equal, so we continue with the loop.
5. Since `ty > tx`, we perform `ty %= tx`.
6. After applying the modulo operation, `ty` is now `4 % 1 = 0`. The target point is now `(1, 0)`.
7. Comparing the new `ty` with `sy`, we see that `ty < sy`; we cannot reduce `ty` anymore without going below `sy`.

From the example above, we note the following:

- After several steps, we've ended up with `ty < sy`, which violates the condition that it's only possible to either match the starting point exactly (in which case both `tx` and `ty` would match `sx` and `sy`) or reduce one of the target coordinates to match the starting point's while checking if the other can achieve the same via multiples.
- Since we could not achieve a situation where `tx` or `ty` matches `sx` or `sy` respectively without the other target coordinate going below its starting counterpart, it proves that the path from `(sx, sy)` to `(tx, ty)` cannot be achieved with the given operations.
- As a result, we would return `false` for this example.

Key Takeaways:

- The reverse approach significantly simplifies the problem, turning an otherwise computationally expensive search into a series of simple subtraction operations (modulo operation).
- If at any point during the reverse operation one of the target coordinates becomes less than its corresponding start coordinate, we can stop and return `false` because this means that the target point cannot be reached from the starting point.

Python Solution

```python
class Solution:
    def reachingPoints(self, start_x: int, start_y: int, target_x: int, target_y: int) -> bool:
        # Loop to transform the target point back towards the starting point
        while target_x > start_x and target_y > start_y and target_x != target_y:
            # If target_x is greater than target_y, reduce target_x
            if target_x > target_y:
                # The modulo operation finds how many steps can be taken from target_y to reach current target_x
                target_x %= target_y
            # If target_y is greater than target_x, reduce target_y
            else:
                # Likewise, this modulo operation finds the steps from target_x to reach current target_y
                target_y %= target_x

        # Check if the starting point is reached after breaking out of the loop
        if target_x == start_x and target_y == start_y:
            return True

        # If only target_x matches start_x, check if we can reach the target point
        # by repeatedly subtracting start_x from target_y
        if target_x == start_x:
            return target_y > start_y and (target_y - start_y) % target_x == 0

        # If only target_y matches start_y, check if we can reach the target point
        # by repeatedly subtracting start_y from target_x
        if target_y == start_y:
            return target_x > start_x and (target_x - start_x) % target_y == 0

        # If neither target_x nor target_y matches, reaching the target point is not possible
        return False

```

Java Solution

```java
class Solution {

    /**
     * Checks if a point (sx, sy) can reach (tx, ty) by moving either vertically or horizontally.
     *
     * @param sx Starting point x-coordinate.
     * @param sy Starting point y-coordinate.
     * @param tx Target point x-coordinate.
     * @param ty Target point y-coordinate.
     * @return True if the starting point can reach the target point, else false.
     */
    public boolean reachingPoints(int startX, int startY, int targetX, int targetY) {
        // Work backwards from the target point to the starting point.
        while (targetX > startX && targetY > startY && targetX != targetY) {
            if (targetX > targetY) {
                // If the current x-coordinate is larger, reduce it modulo the y-coordinate.
                targetX %= targetY;
            } else {
                // If the current y-coordinate is larger or equal, reduce it modulo the x-coordinate.
                targetY %= targetX;
            }
        }

        // If we have reached the starting point, return true.
        if (targetX == startX && targetY == startY) {
            return true;
        }

        // If we are in the same horizontal line as the starting point, check if we can reach by vertical moves.
        if (targetX == startX) {
            return targetY > startY && (targetY - startY) % startX == 0;
        }

        // If we are in the same vertical line as the starting point, check if we can reach by horizontal moves.
        if (targetY == startY) {
            return targetX > startX && (targetX - startX) % targetY == 0;
        }

        // If none of the above conditions are met, the target cannot be reached from the starting point.
        return false;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to determine if we can reach the point (tx, ty) starting at (sx, sy)
    bool reachingPoints(int startX, int startY, int targetX, int targetY) {
        // Run the loop until either of the target co-ordinates is greater than the start co-ordinates
        // and they are not the same. This is because we can move from (x, y) to (x, x+y) or (x+y, y),
        // not the other way around.
        while (targetX > startX && targetY > startY && targetX != targetY) {
            // If targetX is greater than targetY, we know in the last move targetX was changed.
            // So we take modulus to revert the last operation and try the previous state.
            if (targetX > targetY) {
                targetX %= targetY;
            }
            // Similarly, if targetY is greater, we find the previous state of targetY by taking modulus.
            else {
                targetY %= targetX;
            }
        }

        // If we have exactly reached the start point, return true.
        if (targetX == startX && targetY == startY) return true;

        // If the targetX is the same as startX, then we can only reach the target by vertical moves.
        // Thus, check if the difference is a multiple of startX.
        if (targetX == startX) return targetY > startY && (targetY - startY) % targetX == 0;

        // If the targetY is the same as startY, then we can only reach the target by horizontal moves.
        // Thus, check if the difference is a multiple of startY.
        if (targetY == startY) return targetX > startX && (targetX - startX) % targetY == 0;

        // If neither of the above cases, we can't reach the target point with the moves allowed.
        return false;
    }
};

```

Typescript Solution

```typescript
// Function to determine if we can reach the point (targetX, targetY) starting at (startX, startY)
function reachingPoints(startX: number, startY: number, targetX: number, targetY: number): boolean {
    // Run the loop until either of the target coordinates is greater than the start coordinates
    // and they are not the same. This is because we can move from (x, y) to (x, x+y) or (x+y, y),
    // not the other way around.
    while (targetX > startX && targetY > startY && targetX !== targetY) {
        // If targetX is greater than targetY, we know in the last move targetX was changed.
        // So we take modulus to revert the last operation and try the previous state.
        if (targetX > targetY) {
            targetX %= targetY;
        }
        // Similarly, if targetY is greater, we find the previous state of targetY by taking modulus.
        else {
            targetY %= targetX;
        }
    }

    // If we have exactly reached the start point, return true.
    if (targetX === startX && targetY === startY) return true;

    // If the targetX is the same as startX, then we can only reach the target by vertical moves.
    // Thus, check if the difference is a multiple of startX.
    if (targetX === startX) return targetY > startY && (targetY - startY) % startX === 0;

    // If the targetY is the same as startY, then we can only reach the target by horizontal moves.
    // Thus, check if the difference is a multiple of startY.
    if (targetY === startY) return targetX > startX && (targetX - startX) % startY === 0;

    // If neither of the above cases, we can't reach the target point with the moves allowed.
    return false;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the provided code primarily depends on the number of iterations it takes for either `tx` or `ty` to be reduced to either `sx` or `sy`, or for `tx` to equal `ty`. In each iteration, either `tx` or `ty` is reduced by a modulo operation, which takes constant time. However, the number of iterations could be large if `tx` and `ty` are much larger than `sx` and `sy`.

In the worst case, the modulo operation reduces the larger number by an amount proportional to the smaller number. Therefore, the time complexity is logarithmic in relation to the difference between the target and the start coordinates, or more formally, `O(log(max(tx - sx, ty - sy)))`.

Space Complexity

The space complexity of the provided code is `O(1)`. This is because the algorithm only uses a fixed amount of additional space (for variables `tx` and `ty`), and there is no additional space usage that grows with the input size.