Reaching Points

A move consists of taking a point `(x, y)` and transforming it to either `(x, x+y)` or `(x+y, y)`.

Given a starting point `(sx, sy)` and a target point `(tx, ty)`, return `True` if and only if a sequence of moves exists to transform the point `(sx, sy)` to `(tx, ty)`. Otherwise, return `False`.

```
Examples:
Input: sx = 1, sy = 1, tx = 3, ty = 5
Output: True
Explanation:
One series of moves that transforms the starting point to the target is:
(1, 1) -> (1, 2)
(1, 2) -> (3, 2)
(3, 2) -> (3, 5)

Input: sx = 1, sy = 1, tx = 2, ty = 2
Output: False

Input: sx = 1, sy = 1, tx = 1, ty = 1
Output: True
```

Note:

- `sx, sy, tx, ty` will all be integers in the range `[1, 10^9]`.

Solution 1

If `sx,sy` occurs in the path of Euclidean method to get GCD (by subtracting lesser value from greater value) of `tx,ty`, then return true.

To see why this is true, consider how the `tx, ty` could have been formed if `tx > ty`. Let `ax, ay` be the pair in previous step. It cannot be `ax, ax+ay` because both `ax` and `ay` are greater than `0`. So the only other possibility is `ax+ay, ay`. This means `ay = ty` and `ax = tx-ty`. Now we can optimize this subtraction a bit by doing `ax = tx % ty` since we will keep subtracting `ty` from `tx` until `tx > ty`.

One special case we need to handle during this optimization is when `tx=9,ty=3,sx=6, sy=3` which can be covered using the condition `if(sy == ty) return (tx - sx) % ty == 0;`
Similar argument applies for `tx <= ty`.

```
    bool reachingPoints(int sx, int sy, int tx, int ty) {
        while(tx >= sx && ty >= sy){
            if(tx > ty){
                if(sy == ty) return (tx - sx) % ty == 0;
                tx %= ty;
            }else{
                if(sx == tx) return (ty - sy) % tx == 0;
                ty %= tx;
            }
        }

        return false;
    }
```

written by blackshuttle original link here

Solution 2

Basic idea:
If we start from `sx,sy`, it will be hard to find `tx, ty`.
If we start from `tx,ty`, we can find only one path to go back to `sx, sy`.
I cut down one by one at first and I got TLE. So I came up with remainder.

First line:
if 2 target points are still bigger than 2 starting point, we reduce target points.
Second line:
check if we reduce target points to (x, y+kx) or (x+ky, y)

Time complexity
I will say `O(logN)` where `N = max(tx,ty)`.

Python:

```
def reachingPoints(self, sx, sy, tx, ty):
        while sx<tx and sy<ty: tx,ty = tx%ty,ty%tx
        return sx==tx and (ty-sy)%sx==0 or sy==ty and (tx-sx)%sy==0
```

C++:

```
bool reachingPoints(int sx, int sy, int tx, int ty) {
        while (sx<tx and sy<ty) if (tx<ty) ty%=tx; else tx%=ty;
        return sx==tx and (ty-sy)%sx==0 or sy==ty and (tx-sx)%sy==0;
    }
```

Java:

```
public boolean reachingPoints(int sx, int sy, int tx, int ty) {
        while (sx<tx && sy<ty) if (tx<ty) ty%=tx; else tx%=ty;
        return sx==tx && (ty-sy)%sx==0 || sy==ty && (tx-sx)%sy==0;
    }
```

written by lee215 original link here

Solution 3

```
class Solution {
    public boolean reachingPoints(int sx, int sy, int tx, int ty) {
        if (sx == tx && sy == ty) {
            return true;
        } else if (tx == ty || sx > tx || sy > ty) {
            return false;
        } else if (tx > ty) {
            int subtract = Math.max(1, (tx - sx)/ty);
            return reachingPoints(sx, sy, tx - subtract * ty, ty);
        } else {
            int subtract = Math.max(1, (ty - sy)/tx);
            return reachingPoints(sx, sy, tx, ty - subtract * tx);
        }
    }
}
```

written by gges5110 original link here