The function f will generate the next iteration of the Difference function. It works as follows:
Enumerate all permutations of the input which are of length 2
Sort these permutations, so {2,1} becomes {1,2}
Sort the permutations list; put this into the variable 'sorted'
We now have the permutations of {a,b,c} thus: {{a,b},{a,b},{b,c},{b,c},{a,c},{a,c}} so we can take every other one of these to get {a,b},{b,c},{a,c}.
We map the 'subtract' function over these, and take the absolute value, so we have: {abs(a-b),abs(b-c),abs(a-c)} which is the definition of the function.

```
f[list_] :=
 Module[{a, b, c}, With[{sorted = Sort[Sort /@ Permutations[{a, b, c}, {2}]] /.
      {a → list[[1]], b → list[[2]], c → list[[3]]}}, (*sorts them
    so we can delete the duplicates eg abc and cba are the same*)
   Abs[#[[1]] - #[[2]]] & (*difference between the two values*)
    /@ (*mapped over the list:*)
    {sorted[[1]], sorted[[3]], sorted[[5]]} (*takes the 1st,
   3rd and 5th values in the sorted list, which is comprised of:
     ab, ab, bc, bc, ca, ca*)
  ]]
```

A demonstration of how it works, to get what it stabilises to. This example is symbolic; it will never terminate because a, b and c will never stop changing; we will keep on subtracting for ever because *Mathematica* doesn't know when the output reaches zero and thus the subtractions stop. We sort the output in order that we don't keep oscillating between {0,1,0) and {0,0,1}, for example.

```
FixedPoint[Sort[f[#]] &, {a, b, c}]
(*this is the way we get what it stabilises to,
but don't execute this as it will never stop!*)
```

The 'real-life' example given in the problem:

```
FixedPoint[Sort[f[#]] &, {15, 39, 8}]
```

```
{0, 1, 1}
```

To crack the bigger problem, define a modified GCD function that can cope with GCD[0, a]; in this case, we take GCD[0,a]==a.

```
defGCD[a_, b_] := If[a == 0 || b == 0, Max[a, b], GCD[a, b]]
```

This is the form of f that I have worked out. It works as follows:
It is called with three variables; we define three local ones that contain the function's parameters but sorted such that a ≥ b ≥ c.
We then move into the Which statement; this catches the case that any a == any b, when we break with the already-proven Abs[a-c].
If the parameters are not of the form {a,a,b}, we then move into the True block of the Which statement, which contains a procedure, here described line by line.
If a-c is coprime to b-c, return 1,
else define a local variable ans = (a-b) mod (b-c)
If ans == 0, return b-c
else if ans > (b-c)/2, return b-c-ans (since the divisors are 'mirrored' across half of (b-c), for example with {a,8,2} as a changes from 8 to 14, we get 6,1,2,3,2,1,6 corresponding to 0,1,2,3,4,5,6
else return our calculated value of ans, (a-b) mod (b-c)
end procedure

Then we define another function which can interpret being called with a list: we want derivedF[a,b,c] to be the same as derivedF[{a,b,c}].

```
derivedF[ai_, bi_, ci_] := Module[{a, b, c, ans},
  (With[{w = Sort[{ai, bi, ci}]}, a = w[[3]]; b = w[[2]]; c = w[[1]];];
   Which[a == b, Abs[a - c],
    a == c, Abs[a - b],
    b == c, Abs[b - a],
    True, (
          If[GCD[a - c, b - c] == 1, 1,
           ans = Mod[a - b, b - c];
           Which[ans == 0, b - c,
            ans > (b - c) / 2, b - c - ans,
            True, ans]]
          )
   ]
  )
 ]

derivedF[l_List] := derivedF[l[[1]], l[[2]], l[[3]]]
 (*split up the list if it's called with a list; now d[a,b,c]==d[{a,b,c}]*)
```

Here's where *Mathematica* really comes into its own; we can use Manipulate to compare the function output easily for a range of inputs. Here, we Manipulate over the range of $1 \leq a,b,c \leq 20$ - this is all the third line of the function specifies. The more interesting line is line 2, which returns a list of values: the function returned by the derived function and the actual calculated value. This was how I tested my hypotheses, and how I checked easily whether my derived version was right (simply add Equal@@ before the first curly brace to get a True: this is right, or a False: this derived value is wrong.

```
Manipulate[
 {derivedF[Sort[{a, b, c}, Less]], FixedPoint[Sort[f[#]] &, {a, b, c}][[2]]},
 {{a, 1, Dynamic[a]}, 1, 20, 1},
 {{b, 1, Dynamic[b]}, 1, 20, 1}, {{c, 1, Dynamic[c]}, 1, 20, 1}]
```