Euclidean algorithm

Goal: find the `gcd` (greatest common divisor) of two integers `a` and `b`.

Example:

```
421 = 111 x 3 + 88
111 =  88 x 1 + 23
 88 =  23 x 3 + 19
 23 =  19 x 1 +  4
 19 =   4 x 4 +  3
  4 =   3 x 1 +  1
  3 =   1 x 3 +  0
```

The last non-zero remainder is 1. This is `gcd(421,111).` Hint: 421 is on this list.

Three similar Python implementations:

```
if b > a:
    a,b = b,a

def gcd(a,b):
    while True:
        m = a % b
        if m == 0:
            return b
        a,b = b,m

print gcd(421,111)  # 1
print gcd(60,24)    # 12
print gcd(11838*2888, 99991987*2888) # 2888
```

The `while True` isn't strictly necessary:

```python
# requires a > b
def gcd(a,b):
    m = a % b
    while m != 0:
        a,b = b,m
        m = a % b
    return b
```

Recursive version:

```python
def gcd(a,b):
    m = a % b
    if m == 0:
        return b
    return gcd(b,m)
```

Explanation

Suppose we have two integers `a` and `b` and we do `m = a mod b` and `m` is not zero.

The idea is that if `a` and `b` have a common divisor, which we seek, then *so does* `m`.

The mod operation can be expressed as

```
m = a - nb
```

where `nb < a` but `(n+1)b > a`. (If there were an integer `n` so that `nb = a`, then we would get zero remainder and return `b` as the result).

Suppose `a` and `b` have a common factor `f`. We can factor `f` from each term of the previous equation:

```
m = a - nb
m = f(a/f - nb/f)
```

leaving integer terms inside the parentheses. But clearly `m` is also evently divided by `f`.

Thus the problem is reduced, because now we can just find `gcd(b,m)`, because `b` and `m` also have the common factor `f`, and the same logic applies.

Finding the multiplicative inverse

Suppose we know `e` and want to find `d` such that `ed mod p = 1`.

If there exists such a `d` then

```
ed mod p = 1
ed mod p + np mod p = 1
```

and because `np mod p = 0`:

```
(ed + np) mod p = 1
```

We will use this fact in a bit.

link

working through Euclid's theorem

Consider `gcd(81,57)`

```
81 = 1(57) + 24
57 = 2(24) +  9
24 = 2(9)  +  6
 9 = 1(6)  +  3
 6 = 2(3)  +  0
```

So `gcd(81,57) = 3`. What we do next is to find integers (one negative) such that

```
p(a) + s(b) = 3
p(81) + s(57) = 3
```

Rearrange the next to last line (line no. 4) from the `gcd` calculation:

```
3 = 9 - 1(6)
```

Substitute for `6` from the line no. 3

```
6 = 24 - 2(9)

3 = 9 - 1[24 - 2(9)]
  = 3(9) - 1(24)
```

Substitute for `9` from line no. 2

```
9 = 57 - 2(24)

3 = 3[57 - 2(24)] - 1(24)
  = 3(57) - 7(24)
```

Substitute for `24` from line no. 1

```
24 = 81 - 1(57)

3 = 3(57) - 7(24)
  = 3(57) - 7[81 - 1(57)]
  = 10(57) - 7(81)
  = -7(81) + 10(57)
```

Thus, we've shown that

```
3 = p(a) + s(b)
```

where `p = -7` and `s = 10`.

But this means that

```
3 = 10(57) - 7(81)
```

and what this means is that `10(57) = 3 mod 81`.

Paraphrasing the link:

We want to do arithmetic modulo *n*, and in particular, for division we need to find the inverse of integers mod *n*. For large numbers, this turns out to be a difficult task (and not always possible).

It is known that a number *x* has an inverse mod *n* (i.e., a number *y* so that `xy = 1 mod n`) if and only if `gcd(x, n) = 1`.

Our example above had `gcd = 3`, but we are really interested in cases where `gcd = 1` because then we are guaranteed that an inverse exists. The simplest way to arrange this is the choose *n* prime, because then every integer less than *n* has `gcd = 1` with *n*.

The following simple Python function finds the inverse:

```
# requires a > b
def eea(a,b):
    s, t = 1, 0
    u, v = 0, 1
    while b != 0:
        q = a / b
        a, b = b, a % b

        tmp = s, t
        s = u - (q * s)
        t = v - (q * t)
        (u,v) = tmp
    return u
```

I combine this with a brute-force approach to finding the inverse:

```
# requires a > b
def caveman(m,p):
    n = 2
    while m*n % p != 1:
        n += 1
    return n
```

and use the two functions like so

```
p = 127
for i in range(2,p):
    r = my_eea(p,i)
    if r < 0:  r += p
    print i, caveman(i,p), r
```

Output

```
> python euclidean.py
2 64 64
3 85 85
4 32 32
5 51 51
6 106 106
7 109 109
8 16 16
...
```

The two methods agree and we may also check by doing (for example)

```
5 * 51 = 255, 127 * 2 = 254
6 * 106 = 636, 127 * 5 = 635
```

For each pair we have that `pr mod n = 1`.