Greatest common divisors algorithms

Batincu Daniel, group 931

The purpose of these programs is computing the gcd of 2 numbers. One of the implementations will also work with large numbers. Aside from the implementations, proofs for each method will be given, and each methon will have a step by step explanation.

The Euclidian Algorithm

In order to see how this algorithm works, we need the Remainder Theorem

Remainder Theorem For all integers a, b with $b > 0$, there is an pair of integers q, called quotient, and r, called remainder, such that $a = qb + r$ with $0 \leq r \leq b$.

The greatest common divisor $d = gcd(a, b)$ of 2 positive integers a and b is an integer with the following property: there exist 2 integers u and v such that $d = au + bv$.

We want to compute the gcd of 2 positive integers a and b. If $a = b$, then the $gcd(a, a) = a$. Otherwise, suppose $a > b$. Now, as long as $b > 0$, we compute $gcd(a, b) = gcd(b, amodb)$. When b reaches 0, the value of a is the result we're looking for.

```
<< Euclidean >>=
def euclid(a,b):
    while b > 0:
        print("gcd(" + str(a) + "," + str(b) + ")")
        r = a % b
        a = b
        b = r
    return a
@
```

Why is this working? Let c be a positive integer. If $a \equiv c(\mod b)$, then $b|a - c$, so there is a y such that $a - c = by$, or, $c = a - by$. If d divides both a and b, then it also divides $a - by$. Therefore $a \equiv c(\mod b) => (a, b) = (b, c)$. (1)

Binary GCD Algorithm

This algorithm uses simpler arithmetic operations than the conventional Euclidean algorithm; it replaces division with arithmetic shifts, comparisons, and subtraction.

When trying to find the gcd of 2 numbers a and b, we find ourselves into one of the next situations:

1. $gcd(a, 0) = a$ or $gcd(0, b) = 0$ because everything divides 0 and a (or b, respectively) is the largest number that divides a (or b, respectively)
2. $gcd(2a, 2b) = 2gcd(a, b)$
3. $gcd(2a, b) = gcd(a, b)$ if b is odd, 2 is not a common divisor. Similarly, $gcd(a, 2b) = gcd(a, b)$ if a is odd
4. $gcd(a, b) = gcd(|a - b|, min(a, b))$ if both are odd

```
<< Binary >>=
def bgcd(a,b):
    print("gcd(" + str(a) + "," + str(b) + ")")
    #simple cases
    if a == b:
        return a
    if a == 0:
        return b
    if b == 0:
        return a

    # check for factors of 2
    if a % 2 == 0:                      # if a is even
        if b % 2 == 1:                  # if b is odd
            return bgcd(a//2,b)
        else:                   # both are even
            print("2 * ")
            return 2 * bgcd(a//2,b//2)
    if b % 2 == 0:                      # a is odd, if b is even
        return bgcd(a,b//2)

    # if both are odd
    return bgcd(abs(a-b),min(a,b))
@
```

This algorithm uses the binary representation of the numbers. If the numbers are both odd during a step, an iteration of the Euclidean Algorithm is executed. In other cases, it uses the fact that, if one of the numbers is odd, 2 is not a divisor; if both are even, 2 is a common divisor.

The Euclidean Algorithm (by repeated subtractions)

This represents the steps behind the Euclidean Algorithm. The idea of the algorithm is: $gcd(a,b) = gcd(| a - b |,min(a,b)) and we stop when $a = b$. This is possible because of (1)

```
<< Subtract >>=
def weakeuclid(a,b):
    if a == b:
        return a
    #print("gcd(" + str(a) + "," + str(b) + ")")
    return weakeuclid(abs(a-b),min(a,b))
@
```

As we can see, for large numbers, a lot of subtractions have to be made, which slows down the program by a lot. So, this algorithm is not recommended for large numbers.

```
<<*>>=
<< Euclidean >>
<< Binary >>
<< Subtract >>

from time import time

t00=time()
print(euclid(363636,1515))
t01=time()

t10=time()
print(bgcd(363636,1515))
t11=time()

t20=time()
print(weakeuclid(363636,1515))
t21=time()

print(str(t01-t00))
print(str(t11-t10))
print(str(t21-t20))


print(euclid(2**100 ,2**10 * 15))
print(bgcd(2**100, 2**10 * 15))
#print(weakeuclid(2**100, 2**10 * 15))
#this one gives an error because it's taking too many steps
```

@

After some runs, the Euclidean Algorithm is the fastest of them all, followed by the Binary GCD. The subtraction method is not reliable for large numbers because it takes a lot of steps. The most suitable one for large numbers is The Euclidean Algorithm (using divisions), because it takes the least amount of steps.