# Algorithms for Calculating The Greatest Common Divisor and The Modular Inverse

## Identifying information

Yacine ELHAMER -

## GCD algorithms

The greatest common divisor of two numbers is defined as a number which divides both of these numbers and that is a multiple of every shared divisor of these two numbers.

This number important in implementing many of the cryptographic algorithms we use today, therefore there is a great need for its computing algorithm to be as efficient as possible, which is what we are going to explore in this report; Specifically, the algorithms we will look at are The Euclidean algorithm, as well as The Extended Euclidean Algorithm.

## Euclidean Algorithm:

This algorithm is based primarily on the following rule: "The gcd of two numbers one of which is null, is the other number". Knowing this, and considering that the set of common divisors of *a*, *b*, and *a-b* is the exact same (since the all get factored out in the substraction), we can reformulate the problem of trying to find *gcd(a, b)* to trying to find *gcd(b, a-b)* (where $a > b$); Furthermore, we could continue on reformulating the problem in same manner untill the problem becomes that of computing *gcd(d, 0)*, in which case we have found the gcd and it is equal to d.

Additionally, in order for us to speed things up, we could omit code which checks for the larger number to use with the following iteration (i.e. should the next recrusive call be *gcd(a, a-b)* or *gcd(b, b-a)*, as well as to skip several recursive calls, we could replace the substraction operation with the modulo operation; This is because the old gcd algorithm (the one using substraction) is essentially going to be repeated until all multiples of *b* are removed from *a*, only at which it starts making progress towards nullifying one of the terms, thereby find the gcd.

With this, a final recursive definition of this algorithm could be given as:

$$\gcd(a,b) = \begin{cases} a, & \text{if } b = 0 \\ \gcd(b, a \bmod b), & \text{otherwise.} \end{cases}$$

Figure 1: gcd-algo

An implmentation of this in algorithm in C is as follows:

```c
int gcd(int a, int b) {
    return b != 0 ? gcd(b, a % b) : a;
}
```

The complexity of this algorithm is given by O(log min(a, b)).

**Extended Euclidean Algorithm:**

This algorithm is a modification of the previous algorithm which helps us express the gcd in terms of the numbers *a* and *b*; i.e. it gives us a solution for the following equation:

$$a \cdot x + b \cdot y = \gcd(a, b)$$

Figure 2: gcd-equation

This algorithm works by setting the coefficients *x* and *y* to the values *1* and *0*, and then working its way up the chain of recursive calls by replacing the *a mod b* expression with *a - [a/b] b*, which is equivalent; This means that with each backwards iteration *n+1* the coefficient of *a* (which is *x _ n+1*) is going to become *y_n*, whole the coefficient of *b* (which is *y_n+1*) is going to become *x_n - [a/b]*y_n* . With this we get a general sense of how this algorithm works.

An implementation of this function is as follows:

```c
int extended_gcd(int a, int b, int *x, int *y) {

    if (b == 0) {
        *x = 1;
        *y = 0;
        return a;
    }

    int x1, y1;
    int d = gcd(b, a % b, &x1, &y1);

    *x = y1;
    *y = x1 - y1 * (a / b);

    return d;
}
```

The complexity of this algorithm is O(log n)

**Modular Inverse:**

The modular inverse of a number $a$ with regards to another number $m$ is defined as the number $x$ such that:

$$a \cdot x \equiv 1 \mod m.$$

Figure 3: mod-inverse

One requirement of this is that the numbers $a$ and $m$ have to be coprime, which is why it is often times most interesting to compute the modular inverses with regards to prime numbers.

An efficient way to find the modular inverse of a number would be to use the extended euclidean algorithm discussed previously. This is because since $a$ and $m$ are coprime, this means that *gcd(a, m) = 1*, therefore applying the extended euclidean algorithm gives us the integer coefficients $x$ and $y$ so that:

$$a \cdot x + m \cdot y = 1$$

Figure 4: mod-inverse-equation

This means that the modular inverse in this case is equal to the number $x$ (since - $m$ * $y$ is a multiple of $m$).

An implementation of this algorithm is given in C as follows:

```c
int mod_inverse(int a, int m) {
    int x, y;

    if(extended_gcd(a, m, &x, &y) == 1)
        return x;
    else
        print("no modular inverse!");
}
```