Algorithms - Series Two(vi)

The RSA algorithm

```
# rsa.py - demo of the RSA algorithm

def main():
    n = e = d = 0
    while 1:
        print("""
    1. Set Public Key
    2. Encode
    3. Decode
    0. Quit
    Your choice? """, end = "")
        choice = int(input())
        if not choice:
            return
        if choice == 1:
            n, e, d = set_keys()
        if choice == 2:
            if not n:
                n = int(input("Public Key: "))
                e = int(input("e: "))
            encode(n, e)
        if choice == 3:
            if not d:
                n, e, d = set_keys()
            decode(d, n)

def set_keys():
    """This fuction asks for 2 primes.
    It sets a public key and an encoding number, 'e'."""
    p = int(input("p: "))
    q = int(input("q: "))
    n = p * q
    m = (p - 1) * (q - 1)
    e = get_e(m)
    print("N = ", n, "\ne = ", e)
    d = get_d(e, m)
    while d < 0:
        d += m
    return [n, e, d]

def get_e(m):
    """Finds an e coprime with m."""
    e = 2
    while gcd(e, m) != 1:
        e += 1
    return e

def get_d(e, m):
    """Takes encoding number, 'e' and the value for 'm' (p-1) * (q-1).
    Returns a decoding number."""
    x = lasty = 0
    lastx = y = 1
    while m != 0:
        q = e // m
        e, m = m, e % m
        x, lastx = lastx - q*x, x
        y, lasty = lasty - q*y, y
    return lastx
```

This is just a simple menu.

The user gets the choice of entering the public (encoding) keys or setting a full set of keys.

Here, we take in two primes (p & q) and use them to set the public key (‘n’), the encoding number (‘e’), the decoding number (‘d’) and the secret key (‘m’).
As this sometimes results in a negative value tfor d, the while loop at the end adds m until we have a positive value.

The encoding number ‘e’ can be a small number, it has to be coprime with ‘m’.

The decoding number, ‘d’ has to satisfy this condition:

```
de % m = 1
```

To get this number, I have used Euclid’s Extended Algorithm (which returns the mutiplicative inverse of e % m).

```
def encode(n, e):
    """This function asks for a number and encodes it using 'n' and 'e'."""
    while 1:
        c = int(input("Number to encode: "))
        if not c:
            return
        print(pow(c, e, n))
```

To encode, we simply raise the user's number, 'c' to the power 'e' and then find the modulus of that number and 'n' - the public key.

```
def decode(d, n):
    """This function asks for a number and decodes it using 'd' and 'n'."""
    while 1:
        c = int(input("Number to decode: "))
        if not c:
            return
        else:
            print(pow(c, d, n))
```

To decode, we raise c to the power of the decoding number 'd' and return the modulus of that value and 'n'.

```
def even(x):
    """True if x is even."""
    return x % 2 == 0

def gcd(a,b):
    """Euclid's Algorithm: Takes two integers and returns gcd."""
    while b > 0:
        a, b = b, a % b
    return a

if __name__ == "__main__":
    main()
```

To use the algorithm, we need values of p and q which are larger than the number we are going to encode. Real-world RSA implementations use really large primes (100 digits is fairly standard!). For our purposes, smaller values will be easier to work with (although obviously not as secure).

Let's say we want to encode a birthday. For this, 9 digit primes will be big enough. Using the Miller-Rabin program, I can find some primes:

```
? 200000000, 1
[200000033]
? 500000000, 1
[500000003]
```

I can use these to generate a public key and an encoding number.

```
p: 200000033
q: 500000003
N =  100000017100000099
e =  3
```

Using these to encode my date of birth (15th June 1970):

```
Number to encode: 15061970
2216315703990170
```

And finally I can decode:

```
Number to decode: 2216315703990170
15061970
```

The security of the RSA algorithm depends upon the fact that it is very difficult to quickly find the prime factors of the public key, ‘n’, if p and q are sufficiently big.

This is how websites keep your credit card information safe online!