TP2 – RSA

Exercice 1 :

1/

(a)

In the following capture you will see two implementations for totient function. The first one I did myself, and the second one using sympy library.

```
# Use totient() method for Euler totient function or phi(n)
#I did this before realising there is the library simpy that does it in one line
#L = []
#for n in range(1, 21):
#    if gcd(n, 20) == 1:
#        L.append(n)

totient_n = totient(n1)
print("phi(n1) = ", totient_n)
```

You can see results by using sage with my python code (‘../SageMath/sage ./tp2-rsa_material.sage.py’ for example).

(b)

As for the previous question, I used the sympy library.

```
#Recover private Key
privateK = mod_inverse(pow(2, 16)+1, totient_n) #this mod_inverse also use Extended Euclid Algo
print('privateK for n1:', privateK)
```

(c)

```
#Recover message
print("Plain text for c1:",int_to_ascii(power_mod(c1, privateK, n1)))
```

In order to recover plaintext from C1, I used previous computation (private Key), a sage function (power_mod) and another one to translate int into ascii caracters. After decrypting, the plaintext is : ‘There is a reason we use two primes and not just one !’

2/

```python
# This function generate prime factors
def pollard(n):
    # defining base
    a = 2
    # defining exponent
    i = 2

    # iterate till a prime factor is obtained
    while(True):
        # recomputing a as required
        a = (a**i) % n

        # finding gcd of a-1 and n using math function
        d = math.gcd((a-1), n)

        # check if factor obtained
        if (d>1 and d<n):
            #return the factor
            return d
        # else increase exponent by one for next round
        i += 1
```

```python
def test_pollard(n):
    # temporarily storing n2
    num = n

    # list for storing prime factors
    rop = []

    # iterated till all prime factors are obtained
    while(True):
        # function call
        d = pollard(num)

        # factor to list
        rop.append(d)

        # reduce num
        r = int(num/d)

        # check for prime using sympy
        if(isprime(r)):
            # both prime factors obtained
            rop.append(r)
            break
        # reduced num is not prime, so repeat
        else:
            num = r
    return rop
```

In order to implement the pollard p-1 function, I used two functions. One for pollard p-1 itself which iterates until it finds a suitable prime, and the second one to test and find the second prime. At the end, we have at least two primes for one given ‘n’. These primes are such that p*p’=n.

3/

After decrypting C2, we get : « In practice, we often choose safe-primes : p=2*p’ + 1, with p’ prime. This ensure tat p-1 has a big prime factor, and avoid such problems ».

Exercice 2 :

Here we use an LSB oracle server that returns the last significant bit of a plaintext p corresponding to a cyphertext c (0 or 1). Knowing that c=p^e[n] and that n is composed of two primes, the goal is locate the interval for p from all returned parity bits.

If $2p < n$ the modulo doesn't come into play and the result is even.

If $2p > n$ the remainder will be odd because n is odd.

For example, If you send the cyphertext of 2p and the result is odd, you'll know that the interval must be within (n/2)<p<n . Now restarts the process in order to further restrict the interval for p. After a few steps, we can get back the plaintext. This method is similar to the dichotomic method and allow us to fully recover p in log2(n) steps.