TP 3 – Chiffrement RSA – Correction

1) On a $n = p \times q = 5 \times 11 = 55$, $\phi(n) = (p-1) \times (q-1) = 4 \times 10 = 40$. L'algorithme d'Euclide pour le calcul du pgcd de e et $\phi(n)$ donne

$$40 = 13 \times 3 + 1.$$

On a donc $\text{pgcd}(e, \phi(n)) = 1$ puis $-13 \times 3 \equiv 1 \mod 40$ et donc $27 \times 3 \equiv 1 \mod 40$. On obtient donc $d = 27$. Ainsi la clé publique est $(55, 3)$ et la clé privée est $(55, 27)$. Le chiffré de $M = 13$ est

$$C = M^e = 13^3 \equiv 52 \mod n$$

et le déchiffré de C est

$$D = C^d = 52^{27} \equiv 13 \mod n.$$

On vérifie qu'on a bien $M = D$.

2) En prenant les notations du sujet avec $M \in \{0, ..., n-1\}$ on a

$$C^d \equiv (M^e)^d \equiv M^{ed} \mod n.$$

Par constrution $ed \equiv 1 \mod \phi(n)$. Il existe donc $k \in \mathbb{Z}$ tel que $ed = 1 + k\phi(n)$. D'où

$$C^d \equiv M^{1+k\phi(n)} \equiv M \times \left(M^{\phi(n)}\right)^k \mod n.$$

Le groupe des inversibles $(\mathbb{Z}/n\mathbb{Z})^*$ de $\mathbb{Z}/n\mathbb{Z}$ est d'ordre $\phi(n)$. Ainsi si $[M]_n$ (classe de M modulo n) est inversible dans $\mathbb{Z}/n\mathbb{Z}$ alors l'ordre de M est un diviseur de $\phi(n)$ et donc $M^{\phi(n)} \equiv 1 \mod n$. Dans ce cas on obtient

$$C^d \equiv M \times \left(M^{\phi(n)}\right)^k \equiv M \times 1^k \equiv M \mod n.$$

On note que $[M]_n$ est inversible dans $\mathbb{Z}/n\mathbb{Z}$ si et seulement si $\text{pgcd}(M, n) = 1$.

Il nous reste donc à traiter le cas $\text{pgcd}(M, n) \neq 1$. Comme $n = pq$ est un produit de premier. On a alors $\text{pgcd}(M, n) \in \{p, q, pq\}$. Le cas $\text{pgcd}(M, n) = pq$ implique n divise M et donc $M = 0$. Dans ce cas on a $M^{ed} = M = 0$ et donc $D \equiv M \mod n$. Il nous reste donc les cas $\text{pgcd}(M, n) = p$ et $\text{pgcd}(M, n) = q$. Les premiers p et q jouant un rôle symétrique regardons seulement le cas $\text{pgcd}(M, n) = p$. De $M \equiv 0 \mod p$ on obtient $M^{ed} \equiv M \mod p$. Montrons qu'on a $M^{ed} \equiv M \mod q$. On a

$$ed = 1 + k\phi(n) = 1 + k(p-1)(q-1)$$

Comme p divise M et pq ne divise pas M on a $\text{pgcd}(M, q) = 1$ et donc $[M]_q$ est inversible dans $\mathbb{Z}/q\mathbb{Z}$. En particulier l'ordre de $(\mathbb{Z}/q\mathbb{Z})^*$, qui vaut $q-1$ est un multiple de l'ordre de M est donc $M^{q-1} \equiv 1 \mod q$. Ainsi on obtient

$$M^{ed} = M^{1+k(p-1)(q-1)} = M \times \left(M^{q-1}\right)^{k(p-1)} \equiv M \times 1^{k(p-1)} \equiv M \mod q.$$

On a ainsi établit dans le cas $\text{pgcd}(M, n) = p$ qu'on a $M^{ed} \equiv M \mod p$ et $M^{ed} \equiv M \mod q$. Nous allons conclure $M^{ed} \equiv 1 \mod pq$ en utilisant le théorème des restes chinois. L'application

$$\begin{array}{rcl} \eta : \mathbb{Z}/pq\mathbb{Z} & \to & \mathbb{Z}/p\mathbb{Z} \times \mathbb{Z}/q\mathbb{Z} \\ [x]_{pq} & \mapsto & ([x]_p, [x]_q) \end{array}$$

est un isomorphisme d'anneau, elle en particulier injective. Par ce qui précède, on a

$$\eta\left(\left[M^{ed}\right]_n\right) = \left(\left[M^{ed}\right]_p, \left[M^{ed}\right]_q\right) = \left([M]_p, [M]_q\right) = \eta\left([M]_n\right).$$

Comme η est injective on obtient $\left[M^{ed}\right]_n = [M]_n$ et donc $C^d = M^{ed} \equiv M \mod n$.

3) L'algorithme naïf effectue exactement p multiplications.

4) Soit $n \in \mathbb{N}$. Notons expr la fonction `exponentitation_rapide`. Montrons par récurrence sur $p \in \mathbb{N}$:

$$H_p : \text{"}\forall a \in \mathbb{N},\ d = \text{expr}(a, p, n) \text{ vérifie } d \equiv a^p \mod n\text{"}$$

Pour $a \in \mathbb{N}$, $\text{expr}(a,0,n)$ retourne 1 et on a bien $a^0 \equiv 1 \mod n$. Donc H_0 est vraie. Soit $p \in \mathbb{N}$. Supposons H_q vraie pour tout $0 \leqslant q \leqslant p$ et montrons H_{p+1}. Soit $a \in \mathbb{N}$. On note q, b et c les variables définies par $\text{expr}(a,p+1,n)$. On a

$$p+1 = \begin{cases} 2q & \text{si } p \text{ pair} \\ 2q+1 & \text{si } p \text{ impair} \end{cases}, \quad b = \text{expr}(a,q,n) \quad \text{et} \quad c = b^2.$$

Par H_q on a $b \equiv a^q \mod n$ et donc $c \equiv a^{2q} \mod n$. En posant $d = \text{expr}(a,p,n)$, on obtient alors

$$d \equiv \begin{cases} c \equiv a^{2q} \mod n & \text{pour } p+1 = 2q \\ a \times c \equiv a^{2q+1} \mod n & \text{pour } p+1 = 2q+1 \end{cases}$$

et donc $d \equiv a^{p+1} \mod n$ dans tous les cas. On a dnc H_p pour tout $p \in \mathbb{N}$ et donc l'algorithme `exponentiation_rapide` est correct.

5) A chaque appel de `exponentiation_rapide` on fait au plus 2 multiplications. Comme à chaque appel on divise le puissance par 2, il y'a au plus $1+\log_2(p)$ appel à la procédure `exponentiation_rapide`. Le nombre de multiplication est donc en $O(\log_2(p))$.

6) En `Sage` l'algorithme `exponentiation_rapide` est

```
def exponentiation_rapide(a,p,n):
    if p==0:
        return 1
    q=p//2
    b=exponentiation_rapide(a,q,n)
    c=b*b
    if p%2==0: # p est pair
        return c%n
    else:
        return (a*c)%n
```

7) Comme $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$ est un corps, l'anneau $\mathbb{F}_p[X]$ est euclidien. Soit $P(X)$ un polynôme de $\mathbb{F}_p[X]$ de degré $n \geqslant 1$. Supposons que $P(X)$ possède une racine α. En effectuant la division euclidienne de $P(X)$ par $(X-\alpha)$ on obtient $Q(X) \in \mathbb{F}_p[X]$ et $c \in \mathbb{F}_p$ tels que $P(X) = Q(X) \times (X-\alpha) + c$. Comme $P(\alpha) = 0$, on obtient $c = 0$ puis que $X - \alpha$ divise P et donc le degré de Q est $n-1$. Par une récurrence immédiate on obtient alors qu'un polynôme de degré $n \geqslant 0$ sur $\mathbb{F}_p[X]$ à au plus n racines.

8) Soit $n \geqslant 3$ un nombre impair (si n est pair alors il n'est pas premier car divisible par 2). Posons $n = 1 + 2^s \times t$ avec t impair. On dit qu'un entier $a \in \{1,...,n\}$ est un témoin de Miller pour n si la suite

$$a^t, a^{2t}, a^{4t}, ..., a^{2^s t}$$

contient que des 1 ou contient $n-1$ modulo n. On commence par coder une fonction `est_temoin_miller(a,s,t)` qui retourne `True` si `a` est un témoin de Miller pour $n = 1 + 2^s\, t$ et `False` sinon.

```
def est_temoin_Miller(a,s,t,n):
    L=[exponentiation_rapide(a,2**k*t,n) for k in range(s+1)]
    S=set(L) # ensemble des éléments de L
    if len(S)==1 and 1 in S:
        return True
    if n-1 in S:
        return True
    return False
```

La fonction `set(L)` retourne l'ensemble obtenu à partir des éléments de la liste `L`. Ainsi `L` contient que des 1 si et seulement si `S` est le singleton $\{1\}$.

Nous obtenons finalement la fonction effectuant le test de Miller-Rabin

```
def test_Miller_Rabin(n):
    if n%2==0:
        return False
    s=valuation(n-1,2)
    t=(n-1)//2**s
    for k in range(20):
        a=randint(1,n-1)
        if not est_temoin_Miller(a,s,t,n):
            return False
    return True
```

La fonction `randint(a,b)` retournant un entier compris entrea$a$ et a,

9) Sachant que n est composite il y'a une chance sur quatre pour qu'un entier a soit un temoin de Miller. Si on teste 20 entiers différents il y'a donc une chance sur 4^{20} pour que le teste de Miller-Rabin détecte un nombre composite n comme étant pseudo-premier, c'est à dire une chance sur

$$4^{20} = 1\,099\,511\,627\,776 \approx 10^{12}.$$

10) La génération d'une paire de clés pour RSA est essentiellement basée sur la génération de grand nombre premier. Pour cela on tire au hasard un grand nombre entier, on prend le plus petit impair plus grand ou égal et on teste le nombre obtenu avec le test de Miller-Rabin. S'il est premier on a fini sinon on recommence.

```
def genere_premier(k): # genere un premier avec k chiffres en base 10
    a=10**(k-1) # plus petit entier avec k chiffres en base 10
    b=10**k-1 # plus grand entier avec k chiffres en base 10
    while(True):
        n=randint(a,b)
        if test_Miller_Rabin(n):
            return n
```

On obtient alors le code suivant pour la gnérations d'une paire de clé pour RSA.

```
def genere_cles(k): # genere des cles RSA avec des premiers a k chiffres
    p=genere_premier(k)
    q=genere_premier(k)
    if p==q: # pas de chance, on recommence
        return genere_cles(k)
    n=p*q
    phi=(p-1)*(q-1)
    e=genere_premier(5) # 5 chiffres est largment suffisant pour e
    if phi%e==0: # pas de chance e divise phi, on recommence
        return genere_cles(k)
    (t,u,v)=xgcd(e,phi) # u*e + v*phi=t=1
    d=u%phi
    return ((n,e),(n,d)) # (clé publique, clé privée)
```

11) Voici les fonctions pour utiliser pour chiffrer et déchiffrer.

```
def chiffre(M,cle):
    (n,e)=cle
    return exponentiation_rapide(M,e,n)
```

```
def dechiffre(M,cle):
    (n,d)=cle
    return exponentiation_rapide(M,d,n)
```

12) Voici une fonction illustrant un cycle complet pour RSA

```
def demonstration(k): # Avec des premiers à k chiffres
    (cle_publique,cle_privee)=genere_cles(k)
    print("Cle publique :",cle_publique)
    print("Cle prive :",cle_privee)
    n=cle_publique[0]
    M=randint(0,n-1)
    print("Message M :",M)
    C=chiffre(M,cle_publique)
    print("Message chiffré :",C)
    D=dechiffre(C,cle_privee)
    print("Message dechiffré :",D)
```