

Pour faire les exercices en Sage, lire `ISIC.sage` du site du cours :

```
CRYPTO = "http://iml.univ-mrs.fr/~kohel/tch/ISIC_M1/"  
exec(open(get_remote_file(CRYPTO + "ISIC.sage")).read())
```

Faire ces exercices sans ordinateur, avant de vérifier des résultats en Sage.

1. Soit $p = 59$, $a = 2$, et $b = 56$.
 - a. Vérifier que $k = 21$ est la clé privée pour la clé publique ElGamal (p, a, b) .
 - b. Calculer un chiffrement de $m = 7$ avec (p, a, b) . Pourquoi n'est-t-il pas unique ?
 - c. Montrer les étapes pour déchiffrement de votre texte chiffré.

Solution.

- a. Il faut vérifier que $a^{21} \bmod 59 = 56$. Pour cela, on calcule les carrés successifs de a modulo 59 :

$$\begin{aligned}a &= 2 \\a^2 &= 4 \\a^4 &= 16 \\a^8 &= 20 = 256 \bmod 59 \\a^{16} &= 46 = 400 \bmod 59\end{aligned}$$

et on exprime $a^{21} \bmod 59 = (a \cdot a^4 \cdot a^{16}) \bmod 59$:

$$\begin{aligned}(2 \cdot 16 \cdot 46) \bmod 59 &= (16 \cdot 96) \bmod 59 = (16 \cdot 33) \bmod 59 \\&= (8 \cdot 66) \bmod 59 = (8 \cdot 7) \bmod 59 = 56.\end{aligned}$$

- b. Pour le chiffrement, il faut choisir un exposant ℓ , pour lequel on prend $\ell = 13$. Le texte chiffré est $(r, s) = (a^\ell \bmod p, mb^\ell \bmod p)$. En utilisant la méthode 'carré-et-multiplication', on trouve

$$(2^\ell \bmod 59, 56^\ell \bmod 59) = (50, 34),$$

et $2 = 7 \cdot 34 \bmod 59$. Donc le chiffrement est $(r, s) = (50, 2)$.

- c. Le déchiffrement est $m = r^{-k}s \bmod p$. L'inverse de 50 modulo 59 est 13, et $33 = 13^{21} \bmod 59$. Donc $33 = r^{-k} \bmod 59$, et $m = (33 \cdot 2) \bmod 59 = 7$.

2. Les fonctions ci-dessous définissent des algorithmes pour la division euclidienne.

```

def div_mod1(a,b):
    (q,r) = (0,a)
    while r > b:
        q += 1
        r -= b
    return (q,r)

def div_mod2(a,b):
    (q,r) = (0,a)
    n = a.nbits() - b.nbits()
    B = 2^n*b
    while n >= 0:
        q *= 2
        if r >= B:
            q += 1
            r += -B
        B = B//2
        n -= 1
    return (q,r)

```

Comparer les deux fonctions `div_mod1` et `div_mod2` pour division euclidienne. Quel est un meilleur algorithme ?

Solution. L'algorithme `div_mod1` prends temps proportionel à a/b , par comparaison, `div_mod2` prends temps proportionel à $\log_2(a/b)$, alors le deuxième version est meilleur. Nous pouvons démontrer les differences des performances de ces algorithmes en Sage.

```

for n in range(12,16):
    print "n = %s:" % n
    t0 = cputime()
    timeit("div_mod1(2^n-1,2)", number=32)
    print "t1:", cputime(t0)
    t0 = cputime()
    timeit("div_mod2(2^n-1,2)", number=32)
    print "t2:", cputime(t0)

```

Ce calcul montre que le temps pour `div_mod1` double en fonction de n , tandis que le temps pour `div_mod2` croit légèrement :

n = 12 :	n = 13 :	n = 14 :	n = 15 :
t1: 0.12072	t1: 0.232516	t1: 0.463912	t1: 0.911344
t2: 0.00504	t2: 0.005113	t2: 0.005622	t2: 0.005854

3. Soit `div_mod(n,m)` un algorithm pour la division euclidienne, qui retourne (q,r) tel que $n = mq + r$. Alors le pgcd étendu peut être calculer avec la fonction suivante :

```

def pgcde(a,b):
    # Trouver (r,u,v) t.q. r = le plus grand diviseur de a et b
    # est r = ua + vb.
    if a < b:
        (r0, r1) = (b, a)
        (u0, u1) = (0, 1)
        (v0, v1) = (1, 0)
    else:
        (r0, r1) = (a, b)

```

```

(u0, u1) = (1, 0)
(v0, v1) = (0, 1)
while r1 > 0:
    (q1, r2) = div_mod(r0, r1)
    (r0, r1) = (r1, r2)
    (u0, u1) = (u1, u0 - u1*q1)
    (v0, v1) = (v1, v0 - v1*q1)
return (r0, u0, v0)

```

Vérifier que $r_0 = u_0a + v_0b$ à chaque iteration de la boucle `while`. Trouver la suite des valeurs (r_0, u_0, v_0) pour $a = 7$ et $b = 59$.

Solution. Chaque étape du boucle construit une triple (r_2, u_2, v_2) et fait la redéfinition :

$$\begin{pmatrix} r_0 & u_0 & v_0 \\ r_1 & u_1 & v_1 \end{pmatrix} \leftarrow \begin{pmatrix} r_1 & u_1 & v_1 \\ r_2 & u_2 & v_2 \end{pmatrix}.$$

On vérifie que les valeurs initiales sont

$$\begin{pmatrix} r_0 & u_0 & v_0 \\ r_1 & u_1 & v_1 \end{pmatrix} = \begin{pmatrix} b & 0 & 1 \\ a & 1 & 0 \end{pmatrix} \text{ où } \begin{pmatrix} a & 1 & 0 \\ b & 0 & 1 \end{pmatrix}.$$

selon le cas $a < b$ ou $b < a$. En particulier, ils satisfont $r_i = u_i a + v_i b$ pour $i = 0$ et 1. La définition des nouvelles valeurs (r_2, u_2, v_2) est

$$u_2 = u_0 - u_1 q_1 \text{ et } v_2 = v_0 - v_1 q_1,$$

où $r_0 = r_2 + q_1 r_1$. Donc, on a :

$$\begin{aligned} r_2 = r_0 - q_1 r_1 &= (u_0 a + v_0 b) - q_1 (u_1 a + v_1 b) \\ &= (u_0 - q_1 u_1) a + (v_0 - q_1 v_1) b = u_2 a + v_2 b, \end{aligned}$$

qui montre que les triples (r_i, u_i, v_i) continuent à satisfaire la relation $r_i = u_i a + v_i b$. En particulier, pour $(a, b) = (7, 59)$, la suite des valeurs de (r_i, u_i, v_i) est

```

(59, 0, 1)
( 7, 1, 0)
( 3, -8, 1)
( 1, 17, -2)

```

4. On a $1 = ua \bmod b$ si $1 = ua + vb$. Utiliser le résultat de l'exercice précédent pour trouver $y = x^{-1} \bmod 59$, pour $x = 7$.

Solution. Comme $1 = 17 \cdot 7 - 2 \cdot 59$, l'inverse de $x = 7$ est $y = 17$ modulo 59.

5. Comparer les deux fonctions `pow_mod1` et `pow_mod2` ci-dessous pour les puissances x^k modulo un entier n :

```

def pow_mod1(x,k,n):
    # Calculer x^k mod n par
    # un algorithme recursif:
    if k == 0:
        return 1
    elif k == 1:
        return x
    elif k == 2:
        return (x^2).mod(n)
    elif k%2 == 1:
        y = pow_mod1(x,k-1,n)
        return (x*y).mod(n)
    else:
        y = pow_mod1(x,k//2,n)
        return (y^2).mod(n)

def pow_mod2(x,k,n):
    # Calculer x^k mod n par
    # la methode du carre et
    # de la multiplication:
    x1 = 1
    x2 = x
    for i in range(k.nbits()):
        if k%2 == 1:
            x1 = (x1*x2).mod(n)
            x2 = (x2^2).mod(n)
        k = k//2
    return x1

```

Pour $x = 7$ et $k = 21$, trouver $x^k \bmod 59$.

Solution. Les deux algorithmes ont la même complexité (mesure d'efficacité), que nous pouvons vérifier en pratique :

```

n = 200
p = next_prime(2^n)
t0 = cputime()
timeit('pow_mod1(7,2^n-1,p)', number=32)
print "t1:", cputime(t0)

```

t1: 2.916656

```

t0 = cputime()
timeit('pow_mod2(7,2^n-1,p)', number=32)
print "t2:", cputime(t0)

```

t2: 2.810054

Pour une exponent qui est une puissance de 2, les algorithmes sont plus efficace, mais pareil :

```

t0 = cputime()
timeit('pow_mod1(7,2^n,p)', number=32)
print "t1:", cputime(t0)

```

t1: 1.477097

```

t0 = cputime()
timeit('pow_mod2(7,2^n,p)', number=32)
print "t2:", cputime(t0)

```

t2: 1.516352

Pour $7^{21} \bmod 59$, les deux algorithmes trouvent la bonne valeur 57.

6. En donnant $x = 7$ et $y = 5$ trouver k tel que $y = x^k \pmod{59}$.

Solution. Parmi les premières puissances $x^k \pmod{59}$, nous trouvons $x^{10} \pmod{59} = y$ pour $k = 10$:

n	x^n	n	x^n
1	7	6	3
2	49	7	21
3	48	8	29
4	41	9	26
5	1	10	5

En général on peut seulement dire k (s'il existe) est borné par 58.