Cryptographie : un jeu d′enfant ?

KENZOUA Florian 47332 Session 2024

Plan détaillé :

1. **Étude théorique du chriffrement RSA :**
   1. Les méthodes de chiffrement
   2. Le chiffrement RSA
   3. Tests de primalité et grands nombres premiers
2. **Implémentation du code RSA sur Python :**
   1. Chiffrage d′un message numérique
   2. Extension aux messages textuels
3. **Exploitation des failles du RSA :**
   1. Méthode de Coppersmith
   2. Algorithme de réduction LLL
   3. Conclusion sur la faisabilité des attaques

1) Les méthodes de chiffrement :

Chriffrement symétrique et chiffrement asymétrique :

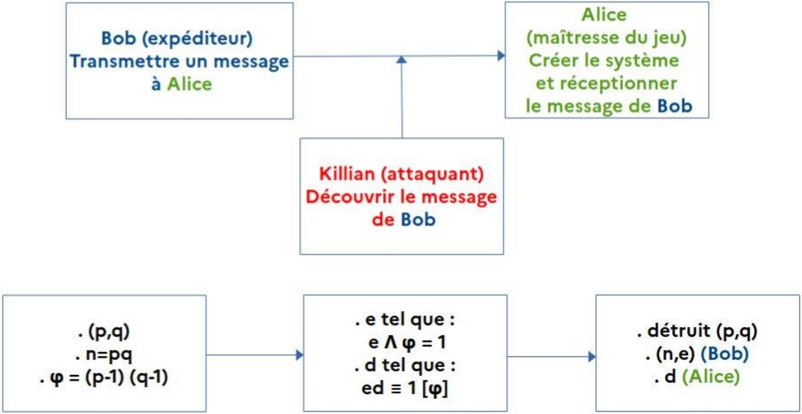
- Chiffrement symétrique :
  - Une clé publique pour chiffrer et pour déchiffrer
- Chiffrement asymétrique :
  - Une clé publique pour chiffrer
  - Une clé privée pour déchiffrer

Intérêt du chiffrement asymétrique :

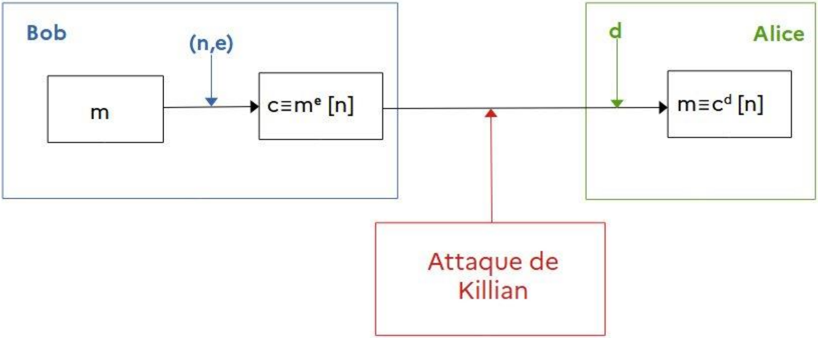
- Mathématiques plus complexes
- Sécurité accrue

2) Le chiffrement RSA :

- Décrit en 1977 par Ronald Rivest, Adi Shamir et Leonard Adleman
- Chiffrement asymétrique (Clé publique et Clé privée)



2) Le chiffrement RSA :



(Annexe 1)

3) Tests de primalité et grands nombres premiers :

Comment Alice peut-elle déterminer son couple de grands nombres premiers (p,q) ?

Pour cela, on peut utiliser :

- Bibliothèque random de Python
- Exponentiation rapide (Annexe 2)
- Test de Miller-Rabin

```python
# Miller-Rabin trial :

def miller_rabin(n, k) :
    if n <= 3 :
        return n == 2 or n ==3
    if n % 2 == 0 :
        return False
    for i in range(k) :
        a = random.randint(2, n - 2)
        if cont_trial(a, n) :
            return False
    return True
```

```python
# One prime number generator :

def prime_num(bits, k) :
    while True :
        n = random.getrandbits(bits)
        n |= (1 << (bits-1)) | 1
        if miller_rabin(n, k) :
            return n

# Two prime numbers generator :

def prime_nums(bits) :
    p = prime_num(bits, 200) # Choose k
    q = prime_num(bits, 200) # Choose k
    while p == q :
        q = prime_nums
    return p, q
```

1) Chiffrage d'un message numérique :

```python
def generate_keys(bits) :
    p, q = prime_nums(bits)
    n = p*q
    phi_n = (p - 1)*(q - 1)
    while True :
        e = random.randint(2, phi_n)
        r, u, v = extended_euclidean(e, phi_n)
        if r == 1 :
            e = e
            d = u % phi_n
            return e, d, n

```

```python
# Encoding the message :

def encode(m, e, n) :
    c = fast_exp(m, e, n)
    return c

# Decoding the message :

def decode(c, d, n) :
    m = fast_exp(c, d, n)
    return m

```

Algorithme d'Euclide (Annexe 3)

Les fonctions permettent :

- Générer les clés (n,e) et d
- Bob : Chiffrer le message numérique m en c tel que :

$$c \equiv m^e \mod n$$

- Alice : Déchiffrer le message c en m tel que :

$$m \equiv c^d \mod n$$

1) Chiffrage d′un message numérique :

```
>>> (executing file "<tmp 8>")
Message :  1234567891011121314151617181920


Public key n :  68979282533811634785663388269625506477232514857220113896535740071143298328841037904354097133315758460 2
9154456309680165285052947133778013097412854306314716815146770650219941453810817068508457368410708064372182347728827406
0458386082687873148437799763929631563508354952109045863229973198329491420087089138851877422122488505076562029195807132
7488100837093980666018246449911448778351542747113347866299513006160520783903656191377510497519939359578049188361140110


Public key e :  19110401404852763617960852822522926368308219508117707713935989103841623203437313569850277505942784297 2
8526285178839103405609268260330785775766239834265858467615506316318990664709435685966095901464852757760584345527779717
9952272258134279720664501888339011026852849953922665210340395886909825569016448879826984347199109011184329368507124220
4118682063623168104592275450156103735875849412609796342027304554759000184185858019558604199088447221566370173475803220


Private key d :  260796837331923997731782061886727023731999493767864797048570710704153732158639068926568222766882420 99
7159033636884920554415966412740501213239727924763770735034196546642119285337257506494016613346531511556354913065902032
1715363127163607434388413467289826043527874246488619733130869489596319853173946320975109869794655594739289169768397868
5612332423342964622540118169591527677706654735433076779636118466277037745100623496640156238708498797382367512295146305


Encoded message :  660130008014904483911945419019249025297837327290151295532068687951458386544320967027369452493875234
9808609063708147125884504112477929074985093611334145400039646591327365965266577521425633699326631480621050765399495891
0596847189671206829718493552291301662600950647680327989367404380949715210564575321171441600830455719541744392767357142
0376682210216769780427952454817130119229966488103532180542752362146388008759204290252556710718479200827719921392757549
08


Decoded message :  1234567891011121314151617181920
```

2) Extension aux messages textuels :

On génère un dictionnaire ASCII et le dictionnaire réciproque :

```python
def generate_dico(n) :
    dico = {}
    for i in range(n+1):
        if i >= 32 and i != 39 :
            dico[i] = len8(Ent2Bin(i))
    return dico

ASCII = generate_dico(255)
```

```python
def reverse_dico(dico) :
    reversed_dico = {}
    for i in range(32, 256) :
        if i != 39 :
            reversed_dico[ASCII[i]] = i
    return reversed_dico

reversed_ASCII = reverse_dico(ASCII)
```

<u>**On traduit ensuite :**</u>

- Texte en binaire (Annexe 4.1)
- Binaire en texte (Annexe 4.2)

Bob et Alice peuvent désormais échanger des messages textuels

2) Extention aux messages textuels :

```
Message :  Alice, voici mon message.


Public key n :  55449119358979733107387968525913781856328279577741943295492676231327724517548249557482945568757137371550899573810
98324707661877550254167036633138787232961839688699491707399148093242274322511323808403339354312205756287465172310525854250241177 2
126668691661864569848629748521218146217776548581800696250772906405518434066420331095065552745861694918669344159818411645739044877
165064095450008369803937974918994915994618502623323411380026795312027614925663595430473471959708711321680639615702331823424793097
4235314933373811017175538627647957594131624933590428912655956513658839639723608255619333532021955659067276114850498530 89903


Public key e :  17667877720145437847655083557071207470867738914402336242582439320066385040557818105638331568055004686414170443648
224554475330853594985449754715601911111737126810421306413819948607588512689644487417774146508011259487485140400821215207628676105
337834752743712480929340999380270058141662741456453377622439734864541869119147339646536313190096960223703901289658109361687988325
618474118450037352234918499629560205271734110878523453566152901511988452774405599883748458358816826858704571234718374761390298417
316096075301539776687147182395426640846253567103454256147611661930657590501734830415455237700464670644160274255146019311821


Private key d :  1299969997315990339872984250338509899385728926549671701953867993672820384928189237660261689363643324160517531940
968935111960218830588212203284083677339590663882343851010648862581761777103875764576196694276687589223214269194673365226282323669
518870641349343567160561073088304943899340101539557814180412776111849756943861732540856092638568896996002035847398769136978961071
347944673658147005184339337327217068542932778555091131379208150044656243528686767998483622259940034081141596030570993200524084707
5800043203886917384257066348469473631339747857546255504674342278204476090368540042962787796709517328794709825344876090686485


Encoded message :  16823698794720775632978666033063790590484744205198375505390295105455970462686002025966210405445072911790279587
980771614980344753121892225853789832383221141970785019115102049336538608925904798725715754535208692927866085895649988129778306528
768476845869588099423398500198490644384505143987829616007049091299623475431065916502219601732718559321966561974074656740790814719
541817370891821912138563684598236469204388765721485160713815562738807339693718782285893944554601694696491943845289456248379075097
378642651476944442725759284675079119276048637226759562990338853413766904786373839629298813549819039817129572768649034895732523


Decoded message :  Alice, voici mon message.
```

1) Méthode de Coppersmith :

L'attaque de Killian :

- La clé publique (n,e)
- Une partie du message b telle que : $m = b + x$ (x inconnu)
- $p(x) = (b + x)^e - c = 0 \mod n$

Méthode de Coppersmith :

- Résoudre le problème => Résoudre équation [n]
- Résoudre équation [n] => Chercher racines dans $\mathbb{Z}$ d'un polynôme

1) Méthode de Coppersmith :

Soit $P \in \mathbb{Z}[x]$ unitaire, $(a_0, ..., a_d) \in \mathbb{Z}$ et $X \in \mathbb{N}^*$ tel que :

- $deg(P) = d \in \mathbb{N}$
- Pour tout $x \in \mathbb{C}$, $P(x) = \sum_{k=0}^{d} a_k x^k$
- $\exists x_0 \in \mathbb{Z},\ x_0 < X,\ P(x_0) = 0 \mod n$

On définit pour tout $R \in \mathbb{K}[x]$ tel que $R(x) = \sum_{k=0}^{d} q_k x^k$ le vecteur $v_R = (q_0, q_1 x, ..., q_d x^d)$

Théorème de Howgrave-Graham (Annexe 5) : Si $x_0 < X$ **est solution de** $P(x) = 0 \mod n$ **avec** $||v_P|| < \frac{n}{\sqrt{d+1}}$ **alors** $P(x_0) = 0$

2) Algorithme de réduction LLL :

Killian veut appliquer ce théorème à un polynôme engendré par la base réduite de : $F = (v_{G_0}, ..., v_{G_{d-1}}, v_P)$ où $(G_i(x))_{0\leq i\leq d-1} = (nx^i)_{0\leq i\leq d-1}$

Gram-Schmidt jusqu'à obtenir :

- $\forall (i,j) \in [|1,m|],\ i < j,\ |\mu_{i,j}| \leq \frac{1}{2}$
- $\forall i \in [|2,m|],\ ||b_i^* + \mu_{i,i-1} b_{i-1}^*||^2 \geq \frac{3}{4}||b_{i-1}^*||^2$

Killian applique donc LLL à F : $(g_1, ..., g_{d+1})$
On note G le polynôme associé à g_1

Howgrave-Graham appliqué à G (Annexe 6) : Si $X < \frac{1}{\sqrt{2}(d+1)^{\frac{1}{d}}} n^{\frac{2}{d(d+1)}}$ et si $x_0 < X$ est solution de $P(x) = 0 \mod n$ alors x_0 est solution de $G(x) = 0$ sur $\mathbb{Z}$

3) Conclusion sur la faisabilité des attaques :

```
n = 10001
X = 10
poly = Polynomial([-222, 5000, 10, 1])
base_1 = Poly_to_Basis(poly, n, X)
base_1 = olll.reduction(base_1, 0.75)
G = Vect_to_Poly(base_1[0], X)

Polynôme P dont on cherche une racine modulo n = 10001 : P(x) = -222.0 + 5000.0 x + 10.0 x**2 + 1.0 x**3


Polynôme G engendré par notre base réduite : G(x) = 444.0 + 1.0 x - 20.0 x**2 - 2.0 x**3


Tableau des racines de G : Rac(G) = [-7.-2.54950976j -7.+2.54950976j  4.+0.j        ]


Partie entière réelle de la racine de G et donc de P modulo 10001 qui nous intéresse pour la vérificatio
n à la main : x = 4
```

En effet :

$$P(4) = 4^3 + 10 \times 4^2 + 5000 \times 4 - 222 = 20002$$

Or :

$2 \times n = 2 \times 10001 = 20002$ d'où $P(4) \equiv 0 \mod n$

3) Conclusion sur la faisabilité des attaques :

Méthode de Coppersmith est compliquée à mettre en place
Une autre méthode : **Factorisation de Richard Schroeppel :**

- entier n
- $O(e^{\sqrt{\log n . \log(\log n)}})$

Longueur	Nb. d'opérations	Durée
50	$1.4 \cdot 10^{10}$	3.9 heures
75	$9.0 \cdot 10^{12}$	104 jours
100	$2.3 \cdot 10^{15}$	74 années
200	$1.2 \cdot 10^{23}$	$3.8 \cdot 10^{9}$ années
300	$1.5 \cdot 10^{29}$	$4.9 \cdot 10^{15}$ années
500	$1.3 \cdot 10^{39}$	$4.2 \cdot 10^{25}$ années

Figure – www.lix.polytechnique.fr

Conclusion :

- Le chiffrement RSA est une méthode très réputée en cryptographie
- On a proposé une implémentation du chiffrement RSA sur Python
- Exploiter les failles du RSA est très difficile à mettre en place

Remerciements :

Merci de votre attention.

Annexe 1 :

Démonstration du système RSA :

Soit $m \in \mathbb{N}$ et $(p, q) \in \mathbb{P}^2$
On définit n, ϕ tels que :

- $n = pq$
- $\phi = (p-1)(q-1)$

On prend e et d tels que :

- $\mathrm{pgcd}(e, \phi) = 1$
- $ed \equiv 1 \mod \phi$

Il existe $k \in \mathbb{N}$ tel que $ed = 1 + k\phi$

- Si $p|m$ alors $m^{ed} \equiv m \mod p$
- Sinon $m = m(m^{p-1})^{k(q-1)} \equiv m \mod p$ d'apès le petit théorème de Fermat
- De même $m^{ed} \equiv m \mod q$

Or $\mathrm{pgcd}(p, q) = 1$ donc $m^{ed} \equiv m \mod pq$ d'après le théorème des restes chinois

Annexe 2 :

```python
def fast_exp(m, e, n) :
    result = 1
    base = m % n
    while e > 0 :
        if e % 2 == 1 :
            result = ( result * base ) % n
        base = ( base * base ) % n
        e //= 2
    return result
```

Figure – Exponentiation rapide

Annexe 3 :

```python
def extended_euclidean(a , b) :
    r0, u0, v0 = a, 1, 0
    r1, u1, v1 = b, 0, 1
    while r1 != 0 :
        r2 = r0 % r1
        q2 = r0 // r1
        u, v = u0, v0
        r0, u0, v0 = r1, u1, v1
        r1, u1, v1 = r2, u - q2*u1, v - q2*v1
    return (r0, u0, v0)

```

Figure – Algorithme d'Euclide

Annexe 4 :

```python
def carac_to_bin(carac) :
    code = ord(carac)
    bin = ASCII[code]
    return bin

def text_to_bin(text) :
    n = len(text)
    bin = ''
    for i in range(n) :
        bin += carac_to_bin(text[i])
    return bin

```

Figure – Annexe 4.1 Texte en binaire

```python
def bin_to_carac(bin):
    code = reversed_ASCII[bin]
    carac = chr(code)
    return carac

def bin_to_text(bin) :
    n = len(bin)
    n = int(n/8)
    text = ''
    for i in range(n) :
        carac = bin[i*8: (i*8) + 8]
        text += bin_to_carac(carac)
    return text

```

Figure – Annexe 4.2 Binaire en texte

Annexe 5 :

Démonstration du théorème de Howgrave-Graham :

On rappelle que : $P(x) = \sum_{k=0}^{d} a_k x^k$ et que $x_0 < X$ avec $X \in \mathbb{N}^*$

alors $|P(x_0)| \leq \sum_{k=0}^{d} |a_k| X^k$

ie $|P(x_0)| \leq \sqrt{d+1}\sqrt{\sum_{k=0}^{d} (a_k X^k)^2}$ d'après Cauchy-Schwartz

ie $|P(x_0)| \leq \sqrt{d+1}\ ||v_P|| < n$

D'où :

$$||v_P|| < \frac{n}{\sqrt{d+1}}$$

Annexe 6 :

Démonstration du théorème de Howgrave-Graham appliqué à G :

On rappelle que :

(1) : $\forall (i,j) \in [|1,m|],\ i < j,\ |\mu_{i,j}| \leq \frac{1}{2}$

(2) : $\forall i \in [|2,m|],\ ||b_i^* + \mu_{i,i-1} b_{i-1}^*||^2 \geq \frac{3}{4}||b_{i-1}^*||^2$

Ce qui donne que :

(3) : $\forall (i,j) \in [|1,m|],\ ||b_j||^2 \leq 2^{i-1}||b_i^*||^2$

Puis applique (3) avec j=1 et en passant au produit de i allant de 1 à m :

(4) : $||b_1|| \leq 2^{\frac{(m-1)}{4}} D^{\frac{1}{m}}$ avec $D = \prod_{i=1}^{m} ||b_i^*||$

Or, on peut montrer avec la famille $(G_0, ..., G_{d1}, P)$ que :

$$D = X^{\frac{d(d+1)}{2}} n^d$$

On injecte enfin dans (4) avec m=d+1 et on obtient :

$$X < \frac{1}{\sqrt{2}(d+1)^{\frac{1}{d}}} n^{\frac{2}{d(d+1)}}$$

Annexe 7 :

Détail complet du code :

```
import random
import numpy as np
from numpy.polynomial import Polynomial
import olll

## USUAL ALGORITHM :

#  Fast exponentiation :

def fast_exp(m, e, n) :
    result = 1
    base = m % n
    while e > 0 :
        if e % 2 == 1 :
            result = ( result * base ) % n
        base = ( base * base ) % n
        e //= 2
    return result

# r = pgcd(a, b) and (u, v) as a*u + b*v = r :

def extended_euclidean(a , b) :
    r0, u0, v0 = a, 1, 0
    r1, u1, v1 = b, 0, 1
    while r1 != 0 :
        r2 = r0 % r1
        q2 = r0 // r1
        u, v = u0, v0
        r0, u0, v0 = r1, u1, v1
        r1, u1, v1 = r2, u - q2*u1, v - q2*v1
    return (r0, u0, v0)

```

```python
## PRIME NUMBERS GENERATION :

# Control trial :

def cont_trial(a, n) :
    k = 0
    d = n - 1
    while d% 2 == 0 :
        k += 1
        d //= 2
    x = fast_exp(a, d, n)
    if x == 1 or x == n - 1 :
        return False
    for i in range(k -1) :
        x = (x*x) % n
        if x == n - 1 :
            return False
    return True

# Miller-Rabin trial :

def miller_rabin(n, k) :
    if n <= 3 :
        return n == 2 or n ==3
    if n % 2 == 0 :
        return False
    for i in range(k) :
        a = random.randint(2, n - 2)
        if cont_trial(a, n) :
            return False
    return True
```

```
# One prime number generator :

def prime_num(bits, k) :
    while True :
        n = random.getrandbits(bits)
        n |= (1 << (bits-1)) | 1
        if miller_rabin(n, k) :
            return n

# Two prime numbers generator :

def prime_nums(bits) :
    p = prime_num(bits, 200)
    q = prime_num(bits, 200)
    while p == q :
        q = prime_nums
    return p, q

## RSA KEY GENERATOR :

def generate_keys(bits) : # Génère les clés (n,e) et d du système.
    p, q = prime_nums(bits)
    n = p*q
    phi_n = (p - 1)*(q - 1)
    while True :
        e = random.randint(2, phi_n)
        r, u, v = extended_euclidean(e, phi_n)
        if r == 1 :
            e = e
            d = u % phi_n
            return e, d, n

```

```python
## TRANSLATION TEXT/BIN :

def Ent2Bin(Ent): # Traduit un entier en binaire.
    n = 0
    if Ent < 2:
        return str(Ent)
    else:
        Q = Ent//2
        R = Ent%2
        BinQ = Ent2Bin(Q)
        Bin = BinQ + str(R)
        return Bin

def len8(chaine) : # Complète une chaîne de caractères de taille strictement inférieure à 8.
    n = len(chaine)
    if n <= 8 :
        n = 8 - n
        chaine = n*'0' + chaine
        return chaine

def generate_dico(n) : # Génère notre dictionnaire ASCII.
    dico = {}
    for i in range(n+1):
        if i >= 32 and i != 39 :
            dico[i] = len8(Ent2Bin(i))
    return dico

ASCII = generate_dico(255) # Génère le dictionnaire ASCII ci-dessous.
```

```
ASCII = {32: '00100000', 33: '00100001', 34: '00100010', 35: '00100011', 36:
'00100100',
 37: '00100101', 38: '00100110', 40: '00101000', 41: '00101001', 42: '00101010',
 43: '00101011', 44: '00101100', 45: '00101101', 46: '00101110', 47: '00101111',
 48: '00110000', 49: '00110001', 50: '00110010', 51: '00110011', 52: '00110100',
 53: '00110101', 54: '00110110', 55: '00110111', 56: '00111000', 57: '00111001',
 58: '00111010', 59: '00111011', 60: '00111100', 61: '00111101', 62: '00111110',
 63: '00111111', 64: '01000000', 65: '01000001', 66: '01000010', 67: '01000011',
 68: '01000100', 69: '01000101', 70: '01000110', 71: '01000111', 72: '01001000',
 73: '01001001', 74: '01001010', 75: '01001011', 76: '01001100', 77: '01001101',
 78: '01001110', 79: '01001111', 80: '01010000', 81: '01010001', 82: '01010010',
 83: '01010011', 84: '01010100', 85: '01010101', 86: '01010110', 87: '01010111',
 88: '01011000', 89: '01011001', 90: '01011010', 91: '01011011', 92: '01011100',
 93: '01011101', 94: '01011110', 95: '01011111', 96: '01100000', 97: '01100001',
 98: '01100010', 99: '01100011', 100: '01100100', 101: '01100101', 102: '01100110',
103: '01100111', 104: '01101000', 105: '01101001', 106: '01101010', 107: '01101011',
108: '01101100', 109: '01101101', 110: '01101110', 111: '01101111', 112
: '01110000', 113: '01110001', 114: '01110010', 115: '01110011', 116: '01110100'
, 117: '01110101', 118: '01110110', 119: '01110111', 120: '01111000', 121: '01111001',
122: '01111010', 123: '01111011', 124: '01111100', 125: '01111101', 126:
'01111110', 127: '01111111', 128: '10000000', 129: '10000001', 130: '10000010',
131: '10000011', 132: '10000100', 133: '10000101', 134: '10000110', 135: '10000111',
136: '10001000', 137: '10001001', 138: '10001010', 139: '10001011', 140: '10001100',
141: '10001101', 142: '10001110', 143: '10001111', 144: '10010000', 145: '10010001',
146: '10010010', 147: '10010011', 148: '10010100', 149: '10010101', 150: '10010110',
151: '10010111', 152: '10011000', 153: '10011001', 154: '10011010', 155: '10011011',
156: '10011100', 157: '10011101', 158: '10011110', 159:
 '10011111', 160: '10100000', 161: '10100001', 162: '10100010', 163: '10100011',
 164: '10100100', 165: '10100101', 166: '10100110', 167: '10100111', 168: '10101000',
169: '10101001', 170: '10101010', 171: '10101011', 172: '10101100', 173: '10101101',
174: '10101110', 175: '10101111', 176: '10110000', 177: '10110001', 178: '10110010',
179: '10110011', 180: '10110100', 181: '10110101', 182: '10110110', 183: '10110111',
184: '10111000', 185: '10111001', 186: '10111010', 187: '10111011', 188: '10111100',
189: '10111101', 190: '10111110', 191: '10111111', 192
: '11000000', 193: '11000001', 194: '11000010', 195: '11000011', 196: '11000100'
, 197: '11000101', 198: '11000110', 199: '11000111', 200: '11001000', 201: '11001001',
202: '11001010', 203: '11001011', 204: '11001100', 205: '11001101', 206:
'11001110', 207: '11001111', 208: '11010000', 209: '11010001', 210: '11010010',
211: '11010011', 212: '11010100', 213: '11010101', 214: '11010110', 215: '11010111',
216: '11011000', 217: '11011001', 218: '11011010', 219: '11011011', 220: '11011100',
221: '11011101', 222: '11011110', 223: '11011111', 224: '11100000', 225: '11100001',
226: '11100010', 227: '11100011', 228: '11100100', 229: '11100101', 230: '11100110',
231: '11100111', 232: '11101000', 233: '11101001', 234: '11101010', 235: '11101011',
236: '11101100', 237: '11101101', 238: '11101110', 239:
 '11101111', 240: '11110000', 241: '11110001', 242: '11110010', 243: '11110011',
 244: '11110100', 245: '11110101', 246: '11110110', 247: '11110111', 248: '11111000',
249: '11111001', 250: '11111010', 251: '11111011', 252: '11111100', 253: '11111101',
254: '11111110', 255: '11111111'}
```

```python
def reverse_dico(dico) : # Permet d'inverser un dictionnaire.
    reversed_dico = {}
    for i in range(32, 256) :
        if i != 39 :
            reversed_dico[ASCII[i]] = i
    return reversed_dico

reversed_ASCII = reverse_dico(ASCII) # Génère le dictionnaire pour le décodage.

def carac_to_bin(carac) : # Traduit un caractère en binaire.
    code = ord(carac)
    bin = ASCII[code]
    return bin

def text_to_bin(text) : # Traduit une chaîne de caractères en binaire.
    n = len(text)
    bin = ''
    for i in range(n) :
        bin += carac_to_bin(text[i])
    return bin

def bin_to_carac(bin): # Traduit du binaire en son caractère correspondant.
    code = reversed_ASCII[bin]
    carac = chr(code)
    return carac

def bin_to_text(bin) : # Traduit du binaire en texte.
    n = len(bin)
    n = int(n/8)
    text = ''
    for i in range(n) :
        carac = bin[i*8: (i*8) + 8]
        text += bin_to_carac(carac)
    return text
```

```python
## RSA ENCODING/DECODING :

# Encoding the message :

def encode(m, e, n) :
    return fast_exp(m, e, n)

# Decoding the message :

def decode(c, d, n) :
    return fast_exp(c, d, n)

## SHOW RESULT :

print('CHIFFREMENT RSA :')
print('\n')

message = 'Alice, voici mon message.' # Message à transmettre.
print('Message : ', message)
print("\n")

message_bin = text_to_bin(message) # Passage du message en chaîne de 0 et 1.

if message_bin[0] == '0' : # Correction si premier terme est 0.
    message_bin_copy = '1' + message_bin
else :
    message_bin_copy = message_bin

message_bin_copy = int(message_bin_copy) # Passage du message en un nombre entier.

e, d, n = generate_keys(1048) # Générer les clés.

Keys = (e, d) # Clé privée.

```

```
encoded_message = encode(message_bin_copy, e, n) # Codage.

decoded_message = str(decode(encoded_message, d, n)) # Décodage.

if message_bin[0] == '0' : # Traitement de la correction, extraction du message initial.
    decoded_message_copy = decoded_message[1:]
else :
    decoded_message_copy = decoded_message

decoded_message_copy = bin_to_text(decoded_message_copy) # Passage du binaire au texte.

print('Public key n : ', n)
print("\n")
print('Public key e : ', e)
print("\n")
print('Private key d : ', d)
print("\n")
print('Encoded message : ', encoded_message)
print("\n")
print('Decoded message : ', decoded_message_copy)

## Attaque par méthode de Coppersmith :

print('\n')
print('METHODE DE COPPERSMITH :')

# Méthode de Coppersmith :
```

```python
def Poly_to_Vect(poly, X, size) : # Passage d'un polynôme à un vecteur associé.
    d = poly.degree()
    n = max(size, d + 1)
    vect = np.zeros(n)
    for i in range(n) :
        if i < d + 1 :
            vect[i] = poly.coef[i]*(X**i)
    return vect

def Vect_to_Poly(vect, X) : # Passage d'un vecteur au polynôme associé.
    n = len(vect)
    poly = Polynomial([0])
    for i in range(n) :
        monomial = Polynomial([0]*i + [1])
        poly += (vect[i]/(X**i))*monomial
    return poly

def Poly_to_Basis(poly, N, X) : # Passage d'un polynôme à une base.
    d = poly.degree()
    a = np.zeros((d + 1, d + 1))
    for i in range(d) :
        monomial = Polynomial([0]*i + [1])
        G = N*monomial
        vect = Poly_to_Vect(G, X, d + 1)
        a[i] = vect
    a[d] = Poly_to_Vect(poly, X, d + 1)
    return a

n = 10001
X = 10
poly = Polynomial([-222, 5000, 10, 1]) # Polynôme que l'on veut tester.
print('\n')
print('Polynôme P dont on cherche une racine modulo n = 10001 : P(x) =', poly)

print('\n')
base_1 = Poly_to_Basis(poly, n, X)
base_1 = olll.reduction(base_1, 0.75) # Réduction LLL.
G = Vect_to_Poly(base_1[0], X)
print('Polynôme G engendré par notre base réduite : G(x) =', G)
print('\n')
print('Tableau des racines de G : Rac(G) =', G.roots())
print('\n')
print('Partie entière réelle de la racine de G et donc de P modulo 10001 qui nous intéresse pour la vérification à la main : x =', int(G.roots()[2].real))
```