TP 4 : RSA

Objectifs : Travailler autour de RSA.

Matériel requis :
- le logiciel SAGE (ou tout autre logiciel de calcul en précision illimitée sur les entiers, comme la commande `bc` de Linux ou le langage PYTHON par exemple)

Préambule : Dans tout le TP, n désigne la partie nommée modulus d'une clé RSA. C'est le produit de deux nombres premiers distincts p et q (gardés secrets), tandis que n est rendu public. Les exposants de chiffrement (public) et déchiffrement (privé) sont notés e et d.

1 L'exponentiation rapide modulaire

Question 1. Programmez l'exponentiation rapide en SAGE (autrement dit en PYTHON).

Dans la suite cette fonction est nommée expo_mod.

2 RSA avec SAGE

2.1 Génération d'une paire de clés

2.1.1 Les nombres premiers

La fonction is_prime est un test de primalité qui prouve la primalité ou non du nombre testé. La fonction is_pseudoprime est un test probabiliste de primalité : avec l'option ">0" c'est le test de Miller-Rabin.

```
sage: is_prime(389)
True
sage: is_prime(2000)
False
sage: is_pseudoprime(389,">0")
True
sage: is_pseudoprime(2000,">0")
False
```

Question 2. Testez ces deux fonctions pour des entiers de 300 chiffres.

Les fonctions next_prime et next_probable_prime donne le premier nombre premier qui suit l'entier passé en paramètre.

```
sage: next_prime(7)
11
sage: next_probable_prime(7)
11
```

Question 3. Déterminez le plus petit nombre premier de 300 chiffres.

2.1.2 Nombres premiers d'une taille donnée

Désignons par t la taille en bits de la clé RSA que l'on souhaite. Il nous faut donc trouver deux nombres premiers de taille $t/2$. Une façon d'atteindre cet objectif consiste à choisir deux nombres de $t/2$ bits au hasard et de trouver les nombres premiers qui les suivent.

Question 4. Écrivez une fonction qui engendre un nombre premier choisi au hasard, sa taille (en bits) étant passée en paramètre.

2.1.3 Exposants publics et privés

Les deux exposants public (e) et privé (d) doivent être tels que $ed \equiv 1 \pmod{\varphi(n)}$, où $\varphi(n) = (p-1)(q-1)$.

Question 5. Pourquoi l'exposant public e (ainsi que l'exposant privé d) ne peut-il pas être pair ?

On peut choisir $e = 65537$ comme exposant public puisque e est premier avec $\varphi(n)$ (avec d'autres valeurs de p et q il pourrait arriver que e ne soit pas premier avec $\varphi(n)$; dans ce cas, soit on change la valeur de$e$, soit on change les nombres premiers p et q).

Question 6. En quoi la valeur $e = 65537$ est-elle un choix intéressant pour l'exposant public ?

Il est possible de regrouper le calcul du pgcd de deux entiers et des coefficients de Bezout avec la fonction xgcd qui applique l'algorithme d'Euclide étendu. Cette fonction renvoie un triplet d'entiers dont la première composante est le pgcd, et les deux autres composantes sont les coefficients de Bezout.

```
sage: r = 876875
sage: s = 654654
sage: g, a, b = xgcd(r,s)
sage: g,a, b
(1, 276275, -370056)
sage: a*r + b*s
1
```

Voici un exemple de création d'une paire de clés RSA de taille 2048 bits :

1. Tout d'abord les nombres premiers, et le modulus

```
sage: p = premier_de_taille(1024)
sage: q = premier_de_taille(1024)
sage: n = p * q
sage: n
18895617064234142779442236873949133128423538721856476125435342790028913538547237926013010829845257840093065261005758984050091783949367561820219818619255088222064415815814977264957482636518072317907993007132441099546093516458811784171768189743041279902060203137343790805898768773835844484415722742751402745448562558815356096832567416502233584134082890306669651931468815179819926497358881446396475980207893323252848646329476829948701369626261144338798867964456609010675801619413938816667648327977078785339442920105617869567812764369005947447433506160272047684176646867121407638423680358278329828439429548344220364204501
```

2. Ensuite vérification que l'exposant $e = 65537$ convient

```
sage: phi = (p - 1) * (q - 1)
sage: e = 65537
sage: g, a, b = xgcd(e,phi)
sage: g
1
```

Le pgcd vaut 1, e convient donc. En quoi la question précédente permettait de prévoir ce succès probable au 1er essai ?

3. Calcul de la clé privée et vérification

```
sage: d = a % phi
sage: e*d % phi
1
```

Question 7. Réalisez une fonction qui engendre une paire de clés RSA publique/privée d'une taille donnée en paramètre, l'exposant public étant égal à 65537. Votre fonction doit renvoyée le triplet (n, e, d).

2.2 Chiffrement et déchiffrement

Les messages que l'on peut chiffrer sont les entiers compris entre 0 (inclus) et n (exclu).
Le chiffrement d'un message m s'obtient par le calcul de

$$c = m^e \pmod{n}.$$

Pour le message $m = 1234$, on obtient

```
sage: m = 1234
sage: c = expo_mod(m,e,n)
sage: c
590644395202761322528706129779686537802554766371727753257588978737713188102676339302518480959236696850636336111869158035278367867232404508047519841143799749532045027315920941638383527227928765449686380741667421583382800319571006133802992618476279384576745000080849770600589742206959694369426175011253526996104619174915575091107261242110266765063767090115176742432522964483516284840401433711361990179737120056608054495946271805968869534036559983261503621974869092790414816121675890355081320185346095879782216370492689852452950876452687800431268479688669786025400986193098731736875083567438261110039122976997250040656 6
```

Le déchiffremenr d'un message chiffré c s'obtient par le calcul de

$$c^d \pmod{n}.$$

```
sage: expo_mod(c,d,n)
1234
```

2.3 Sécurité

La sécurité de RSA repose sur la difficulté de factoriser le modulus de la clé publique.
La fonction factor de SAGE factorise les entiers qu'on lui passe en paramètre. Par exemple,

```
sage: factor(252)
2^2 * 3^2 * 7
```

donne la factorisation $252 = 2^2 \times 3^2 \times 7$.
Question 8. Générez au hasard des modulus de clé RSA de taille de plus en plus grande, et constatez l'augmentation du temps de calcul de la factorisation de ce modulus.

3 Utilisation de RSA

Question 9. Récupérez le fichier `les_cryptogrammes4.zip` et décompressez-le.

On utilise RSA essentiellement pour communiquer de manière confidentielle une clé d'un système à clé secrète.
Le fichier `cryptogram16` a été chiffré en utilisant l'AES en mode CBC avec un vecteur d'initialisation nul et une clé de 128 bits. Cette clé K a été elle-même chiffrée avec la clé publique RSA

$$\begin{aligned} n &= 484001516976824291824081505569967425918027658822251613166283 7 \\ e &= 65537, \end{aligned}$$

et on obtient

$$K^e \pmod{n} = 233627333367588510154859814969759518085615053960877783737066 2.$$

Question 10. Le modulus n est un produit de deux nombres premiers assez proches l'un de l'autre. Il est possible de le factoriser par la méthode de Fermat. Faîtes-le.

Question 11. Déduisez-en la clé privée d, puis retrouvez la clé secrète K.

Question 12. Une fois la clé K connue, utilisez `openssl` (cf le TP2) pour déchiffrer le cryptogramme. Que représente l'image obtenue ?

4 Factoriser le modulus

Il existe un algorithme probabiliste qui permet (avec une probabilité non négligeable) de factoriser le modulus n en temps polynomial, lorsqu'on connaît les exposants de chiffrement e et de déchiffrement d.

```
########################################
# Factorisation de n connaissant e et d
########################################

# recherche de l'entier impair u tq ed-1 = u*2^l
u = e*d - 1
while u % 2 == 0:   u //= 2

# recherche d'une racine carree modulo n de 1
# de la forme a^(u*2^m) mod n
# a entier aleatoire dans Z_n

a = ZZ.random_element(2..n)


if gcd(a,n) != 1: return (a,n//a)

b = expo_mod(a,u,n)
b2 = b*b % n
while b2 != 1:
    b = b2
    b2 = b*b % n

p1 := igcd(b-1,n)
q1 := igcd(b+1,n)

# BINGO si 1< p1 <n
# sinon on recommence avec une autre valeur de a
```

Question 13. Programmez cet algorithme probabiliste et testez-le.

5 L'attaque de Wiener

M. Wiener a publié en 1990 (*Cryptanalysis of Short RSA Secret Exponents*, IEEE Transaction on Information Theory, vol 36, n° 3, mai 1990) une attaque qui permet de retrouver la clé privée d en connaissant uniquement la clé publique e et n lorsque cette clé privée d est inférieure à la racine quatrième du modulus n.

Sa technique s'appuie sur le développement en fraction continue de e/n. Parmi les réduites obtenues, l'une a pour dénominateur l'exposant privé d.

Voici un exemple en MAPLE

```
########################################
# Attaque de Wiener
########################################
# la cle publique
sage: n = 7507749913; e = 3217382455
# calcul des reduites de la fraction continue associee a e/n
sage: cvgts = continued_fraction(e/n).convergents()
# Voici les reduites
sage: cvgts
[0,
 1/2,
 2/5,
```

```
 3/7,
 2045/4772,
 6138/14323,
 45011/105033,
 51149/119356,
 3011653/7027681,
 3062802/7147037,
 6074455/14174718,
 9137257/21321755,
 24348969/56818228,
 57835195/134958211,
 82184164/191776439,
 222203523/518511089,
 748794733/1747309706,
 3217382455/7507749913]

# Recherche de la cle privee parmi les denominateurs des reduites
sage: m = 12345
sage: c = expo_mod(m,e,n)
sage: i = 0
sage: while i <= len(cvgts) and expo_mod(c, cvgts[i].denominator(),n) != m:
          i += 1

sage: i, cvgts[i].denominator()
(3, 7)

# BINGO !! on a trouve la cle privee
```

Question 14. Pourquoi pourrait-il être intéressant (pour le titulaire de la clé) d'avoir un petit exposant de déchiffrement ?

Question 15. Les cryptogrammes suivants ont été produits avec des clés RSA dont l'exposant de déchiffrement satisfait l'hypothèse de l'attaque de Wiener.

Chiffré	Modulus	Exposant de chiffrement
783829455698841046080191161 8	726076907963822195338556015 7	677159723405737836471611843 3
474336965523762977662313406 5	422697983935892589478301226 4	029817839390670196399192710 8
592536592528368972073992144 6	746192242312064761807469686 5	027628141059072148144735004 4
106628478365403845720240383 0	560327989977672186524038499 9	194135093898387521607321728 9
044504376648763024360414206 2	278415098285210206719578492 4	746481138039808478820690147 8
997708360896189100881098719 3	553481000061931627100261501 0	376077107638769067644093024 7
983820035454812581609657041 3	904204768191588911074788655 2	896667015586547005213922476 6
755061613485229318236280200 9	100484259456562186576156821 9	182261668833464366482138348 4
048890679686093695681420980 6	736813805495030606420410646 7	032774980431792606135434693 6
242046039613716276965821449 3	489805123379408585387506517 9	254039078032252161732203490 7
917660515828417381339578346	889076860401832811516500557 3	064050933691430165924843906 7

Retrouvez le texte original en utilisant la procédure nombreEnTexte décrite dans l'annexe (cf 7).

6 Produire des clés avec trappe

Il est possible de s'appuyer sur l'attaque de Wiener pour produire des clés RSA avec trappe. Voici un générateur de clés RSA, proposé par C. Crépeau et A. Slakmon, qui permet à son auteur de retrouver la clé privée correspondant à une clé publique produite par ce générateur. Voici décrit ce générateur (tel qu'on le trouve sur leur site) :

```
Let M := a STRENGTH bit even constant fixed in the program.
REPEAT pick a random number P of STRENGTH/2 bits UNTIL it is a prime
REPEAT pick a random number Q of STRENGTH/2 bits UNTIL it is a prime
```

```
Let N:=P × Q, Φ:=(P − 1) × (Q − 1)
REPEAT
 REPEAT
  pick a random number D such that |D| < |N|/4
 UNTIL gcd(D,Φ)=1
 find E such that D × E = 1 (mod Φ).
 let E′ := E + M
UNTIL gcd(E′,Φ) = 1
find D′ such that D′ × E′ = 1 (mod Φ).
Output Private Key := (D′,P,Q), Public Key := (E′,N).
```

où STRENGTH désigne le nombre de bits de la clé à produire, et la notation $|D|$ désigne le nombre de bits de l'entier D.

Question 16. Quel algorithme faut-il utiliser dans ce programme pour calculer E et D' ?

Question 17. La clé privée produite par ce générateur vérifie-t-elle les hypothèses de l'attaque de Wiener ?

Question 18. Indiquez comment l'auteur de ce programme peut retrouver la clé privée en connaissant la clé publique.

Question 19. Retrouvez la partie privée de la clé RSA produite par un tel générateur en connaissant la partie publique, ... et la trappe ! Puis déchiffrez le message.

Modulus	**Exposant de chiffrement**	**Trappe**	**Message chiffré**
152340917674053822933312	125603267856784279105792	186166570161033043410344	150566825798183151272193
481998971581584529236883	380706052139443130474321	415770536751930267094255	845925935647007323690405
325414640426105789290371	632965374679126523178492	468204654566224615606416	746131468305068567581232
333758134502765704181365	596119550217241540705526	501000464581582889085097	376685647560466212707710
460990236623310066275421	673608669228200186826162	059611704946390206585951	305344288524573658357964
508250006903145242509195	225482364635730386515403	127789811172455336262347	989530981388134753958177
070233606095080194282835	712625179764253741641405	614684258490475340792297	712687753429812102651545
412907028509891138620468	000054063831756916855459	861252695166802312474877	033012720613202387992863
878270276627672213131003	932644016615405233011448	110961633292161499439705	922621689569805194988948
270466128386612489928808	673498452095136507943274	000793854819238848003438	315688264114727380545939
837618541574658762281569	033162396876321069107400	801966378788611186180962	419216700437015595721697
053798763614541265975492	464473902591465701212738	381648173603420993167483	123087652799909165018013
749269724529905586066157	663287378935769704372947	173638773944972565402	968670985640212679646282
637862937225736787461034	900773307748415925238570		380125725245915672331529
483829174300674191279786	882144183613521397925073		652422569477076246185400
432438902182552555861193	332003303445429421113163		305993428630511999931698
556282634429744890479291	983486249912977343613940		959439424268386348292065
730190275667359117733408	729362568900254280288240		078272126520604434464823
299473164114354736692242	440773536719244796377086		348274024527373651474523
116917742111680682874704	794966563425342578682925		212852626121164388786911
836901243413246501134302	390875910222210811098223		635634690913295707760041
622581197048403230981457	339750838434773242738397		867693259712122132586387
697234654182126299402343	093826617993518096367769		743748904515420699443118
689511821192973424171138	729120207583507851040869		994175851667456276355321
339664886818605329938329	705847717198462630948617		365137907947989288635778
76155650441484201	56536881923889259		26864208570225975

7 Annexe

Les cryptogrammes des parties 5 et 6 sont obtenus par le codage de petits textes en français transformés en nombres entiers. L'alphabet utilisé pour les textes est constitué des 26 lettres (majuscules et non accentuées) de l'alphabet latin, ainsi que de l'espace. L'alphabet comporte ainsi 27 caractères.

Le codage utilisé pour transformer un texte en un nombre consiste à transformer chaque lettre en un nombre compris entre 10 et 36 et à concaténer les nombres entre eux. Avec ce codage, le texte `PAC` est transformé en le nombre `261113`