TP 8 : Factorisation, tests de primalité, et RSA

1 Factorisation et Crible d'Eratosthène

1.1 Un algorithme de factorisation naïf

Prenons un nombre comme $N = 11\,385$, dont on souhaite calculer la décomposition en facteurs premiers. Comme N n'est pas pair, on va tenter tous les nombres impairs. On commence par $p = 3$. N est divisible deux fois par 3, on obtient $N = 3^2 \times N'$ avec $N' = 1265$. On continue avec l'entier impair suivant (5) et N'. N' s'écrit $5 \times N''$ avec $N'' = 253$. On continue avec les entiers impairs qui suivent, 7 et 9 ne divisent pas N'', contrairement à 11, en effet $N'' = 11 \times N'''$ avec $N''' = 23$. Comme $11^2 > N'''$ le nombre N''' est premier (car sinon il aurait un diviseur premier inférieur à sa racine carrée). On en déduit la décomposition de N en facteurs premiers : $N = 3^2 \times 5 \times 11 \times 23$.

Exercice 1. Écrire une fonction `decomposition(N)` prenant en entrée un entier $N \geq 1$ et retournant une liste de couples (p_i, n_i) tels que les p_i sont des entiers premiers distincts, les n_i des entiers strictement positifs et $N = \prod_i p_i^{n_i}$.

```
>>> decomposition(600)
[(2, 3), (3, 1), (5, 2)]
>>> decomposition(11385)
[(3, 2), (5, 1), (11, 1), (23, 1)]
>>> decomposition(1)
[]
>>> decomposition(65521)
[(65521,1)]
```

Indication : on pourra traiter à part le cas $p = 2$, et ensuite considérer les entiers impairs $p = 3, 5, 7....$ On divisera N par p autant que possible, en stockant les entiers (nécessairement premiers) apparaissant dans la décomposition. On pourra s'arrêter lorsque N est strictement inférieur à p^2 : si $N > 1$, on a alors N premier. Voici un schéma possible :

```
def decomposition(N):
    L=[]
    n=0
    while N%2==0:
        N=N//2
        n+=1
    if n>0:
        L.append((2,n))
    p=3
    while N>=p**2:
        n=0
        while N%p==0:
            [...]
        [...]
    [...]
    if N>1:
        L.append((N,1))
    return L
```

Exercice 2. *Pour la forme.* Écrire une fonction `recomposition(L)` faisant l'inverse.

```
>>> recomposition(decomposition(11385))
11385
```

Complexité. Lorsque l'on analyse le temps d'exécution d'un algorithme arithmétique, on exprime en général sa complexité en fonction de la taille (nombre de bits) de l'entier considéré. La taille de N étant $n \simeq \log_2(N)$, l'algorithme ci-dessus a donc une complexité exponentielle en n (le pire cas étant naturellement quand N est premier). En effet, cette complexité est en $O(N) = O(2^n)$, ou peut-être $O(\sqrt{N}) = O(2^{\frac{n}{2}})$ avec une amélioration, ce qui reste exponentiel. On ne connaît pas d'algorithme ayant une complexité *polynomiale* en n à l'heure actuelle [1], bien qu'il existe des algorithmes plus efficaces.

1. Et il est probable qu'il n'en existe pas... Ce qui est rassurant car la sécurité des protocoles de chiffrement de données les plus utilisés repose sur la difficulté de factoriser un nombre ! C'est le cas du cryptosystème RSA présenté plus loin.

1.2 Crible d'Eratosthène pour générer tous les nombres premiers inférieurs à une certaine borne

L'algorithme du crible d'Eratosthène permet d'obtenir tous les nombres premiers strictement inférieurs à une certaine borne N. L'idée est la suivante :

- on crée une liste `B` de booléens de taille N. À la fin de l'algorithme, on aura `B[i]` = `True` si et seulement si i est premier, pour tout $i \in [\![0, N-1]\!]$.
- 0 et 1 ne sont pas premiers : on positionne `B[0]` et `B[1]` à `False`.
- On démarre ensuite une boucle, prenant toutes les valeurs entre 2 et $N-1$:
  - le premier entier, 2, vérifie `B[2]` = `True`, il est premier : on positionne tous ses multiples stricts (4, 6, 8,...) à `False` ;
  - le suivant, 3, est premier également : on positionne tous ses multiples stricts à `False`. 6 a déja été coché, on peut commencer à $9 = 3^2$;
  - 4 n'est pas premier ;
  - le suivant, 5, est premier : on positionne tous ses multiples stricts à `False`. 10, 15 et 20 ont déja été cochés, on peut commencer à $25 = 5^2$;
  - on poursuit ainsi jusqu'à la fin (on pourrait s'arrêter à $\sqrt{N}$) : lorsqu'un nouveau nombre premier p est trouvé, on « coche » comme non premiers les entiers $p^2, p^2+p, p^2+2p, \ldots$.
- les entiers i tels que `B[i]` = `True` sont les entiers premiers de $[\![0, N-1]\!]$.

Exercice 3. Implémenter l'algorithme d'Eratosthène sous la forme d'une fonction `eratosthene(N)` permettant d'obtenir tous les entiers premiers strictement inférieurs à N.

```
>>> eratosthene(100)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

Rappel : `range(p**2, N, p)` parcourt tous les entiers $p^2, p^2+p, p^2+2p, \ldots$ strictement inférieurs à N.

Complexité. On peut montrer (à l'aide de théorèmes d'arithmétique assez poussés [2]) que la complexité de l'algorithme du crible d'Eratosthène a une complexité $O(N \log\log N)$. Ce résultat montre qu'il est quasiment optimal pour trouver *tous* les nombres premiers inférieurs à une borne. Cet algorithme permet de générer facilement des listes de nombres premiers, qui peuvent être utilisées pour coder un algorithme de factorisation qui explore uniquement les nombres premiers. Dans la suite, on veut décider si *un nombre particulier* est premier.

2 Tests de primalité

Contrairement au problème de la factorisation, il est possible de tester si un nombre N est premier avec une complexité polynomiale en la taille de N, c'est-à-dire en temps $O(\log(N)^\alpha)$ avec un certain α fixé. Ce résultat est connu depuis 2002. Dans cette section, on va se contenter d'écrire un algorithme *probabiliste* de complexité polynomiale, permettant de décider si un nombre est premier avec une probabilité aussi proche de 1 que souhaitée.

2.1 Un algorithme naïf

Exercice 4. Écrire une fonction `est_premier(N)` basique indiquant si N est premier (on pourra utiliser la fonction `decomposition(N)` de la section précédente).

```
>>> est_premier(65521)
True
>>> est_premier(993905641)
False
```

Indication : il suffit que la liste obtenue par `decomposition(N)` soit de taille 1, et que le deuxième élément de son unique couple soit 1.

2. Essentiellement le théorème des nombres premiers, qui affirme que le nombre de nombres premiers inférieurs à N est équivalent à $N/\ln(N)$ lorsque N tend vers l'infini.

2.2 Test de Fermat

On rappelle le petit théorème de Fermat :

Théorème 1. *Soit N un entier premier et a un entier premier avec N alors $a^{N-1} \equiv 1[N]$.*

On va utiliser la contraposée de ce théorème pour décider si N est un entier premier : s'il existe $a \in [\![1, N-1]\!]$ tel que $a^{N-1} \not\equiv 1[N]$, alors N n'est pas premier. Notez que ce n'est pas parce qu'on trouve un tel a qu'on peut factoriser N pour autant : on sait simplement qu'il n'est pas premier.

Exercice 5. Écrire une fonction `exp_mod(x,n,p)` calculant x^n modulo p. On utilisera un algorithme d'exponentiation rapide. Pour mémoire, dans l'algorithme du cours on maintient un invariant de boucle de la forme $zy^m = x^n$, avec initialement $z = 1, y = x$ et $m = n$, et on fait des divisions euclidiennes successives de m par 2. Ici, on fera attention à travailler modulo p : il n'est pas question de calculer x^n pour calculer ensuite le reste dans la division euclidienne par p à la fin : ce serait trop coûteux en temps comme en mémoire. Par exemple (**sauvegardez votre script avant de tester, au cas où vous n'ayez pas fait bien attention à prendre des modulo p partout ! Le résultat est instantané normalement**) :

```
>>> exp_mod(2,10**20,100)
76
```

(Le nombre $2^{10^{20}}$ possède 10^{20} chiffres en binaire, il faut donc douze millions de gigaoctets pour le stocker...)

Exercice 6. En déduire une fonction `est_fermat(N,a)` prenant en entrée un entier N et un entier a supposé premier avec N, et testant si $a^{N-1} \equiv 1[N]$ (votre fonction renvoie un booléen). Donner la liste des nombres inférieurs à 10 000 qui passent le test de Fermat pour $a = 2$ mais qui ne sont pas premiers. (Il y en a 3 inférieurs à 1000, qui sont 341, 561 et 645). Ces nombres sont appelés nombres de Poulet.

```
>>> est_fermat(561,2)
True
```

Exercice 7. Le logiciel de chiffrement PGP décide qu'un nombre supérieur à 8 est premier s'il passe le test de Fermat pour $a = 2, 3, 5$ et 7. Écrire une fonction `est_premier_pgp(N)` retournant un booléen indiquant si N est premier au sens du logiciel PGP. Donnez la liste des nombres inférieurs à 100 000, premiers au sens du logiciel PGP mais non premiers (il y en a 3 !)

```
>>> est_premier_pgp(993905641) #un nombre qui passe le test PGP
True
>>> decomposition(993905641) #mais qui n'est pas premier !
[(41, 1), (71, 1), (421, 1), (811, 1)]
```

Nombre de Carmichael. On appelle *nombre de Carmichael* un nombre non premier qui passe le test de Fermat pour tout $a \in [\![2, N-1]\!]$ qui est premier avec N. C'est le cas de 561 ou de 993 905 641 par exemple. Il a été montré que les nombres de Carmichael sont en nombre infini, mais sont relativement rares par rapport aux nombres premiers. Il en va de même de ceux qui passent simplement le test du chiffrement PGP. Ainsi, pour de grands entiers, un entier déclaré premier par le test PGP a de très grandes chances d'être premier.

2.3 Test de Miller-Rabin

Le test de la section précédente est légèrement problématique pour s'assurer avec certitude qu'un nombre est premier, car il existe des nombres (ceux de Carmichael) pour lesquels $a^{N-1} \equiv 1[N]$ pour tout a premier avec N. On ne peut donc pas prendre beaucoup de a au hasard et s'arrêter dès qu'on en a trouvé un tel que $a^{N-1} \not\equiv 1[N]$ (ou qu'on a testé suffisamment de a). Il faudrait en effet qu'on ait beaucoup de chance pour déclarer un nombre de Carmichael non premier : tomber sur un a qui a un facteur commun avec N est peu probable. Un test probabiliste plus complexe repose sur le résultat de Miller-Rabin :

Théorème 2. *Soit N un nombre premier impair, tel que $N - 1 = 2^s \times m$, avec m impair (s est l'exposant de 2 dans la décomposition de $N-1$ en facteurs premiers). Alors pour tout $a \in [\![1, N-1]\!]$, l'une des deux conditions suivantes est vérifiée :*

— $a^m \equiv 1[N]$

— $\exists d \in [\![0, s-1]\!]$, *vérifiant* $a^{2^d \times m} \equiv -1[N]$

Le théorème n'est pas très dur à montrer à partir du théorème de Fermat, il utilise en plus le fait que si N est premier impair, il n'y a modulo N que deux racines carrés de 1 : 1 et -1 (alors que par exemple modulo 15, il y en a 4 qui sont ± 1 et ± 4).

Ce qui rend ce théorème plus intéressant que le simple théorème de Fermat est le fait que si N n'est pas premier, au plus 1/4 des éléments de $[\![1, N-1]\!]$ vérifient l'une des deux conditions précédentes. On appelle *témoin* de N un entier $a \in [\![1, N-1]\!]$ ne vérifiant aucune des deux conditions précédentes.

Exercice 8. Écrire une fonction `temoin(N,a,m,s)` prenant en entrée N (entier impair ≥ 3), $a \in [\![1, N-1]\!]$ ainsi que les deux entiers m et s tels que $N - 1 = 2^s \times m$ et m impair, et retournant un booléen indiquant si `a` est un témoin pour N. On procèdera en deux temps : d'abord le calcul de a^m modulo N (par exponentiation rapide modulaire), puis élévations au carré successives (modulo N toujours) pour les $a^{m \times 2^d}$ modulo N.

```
>>> temoin(561,2,35,4) #560=2**4*35. a=2 suffit pour voir que 561 n'est pas premier !
True
>>> temoin(561,50,35,4) #avec 50, c'est comme si 561 était premier.
False
```

Le test de Miller-Rabin consiste à choisir des entiers $a \in [\![1, N-1]\!]$ aléatoires, et tester s'ils sont témoins de N. Si on trouve un témoin de N, cela signifie qu'il n'est pas premier. Si suffisamment de tests ont été effectués, l'entier N peut-être déclaré premier car il l'est avec une grande probabilité.

Exercice 9. Écrire la fonction `miller_rabin(N)` permettant de déclarer si un nombre est premier avec grande probabilité. 30 tests de témoins seront suffisants pour déclarer N premier [3].

```
>>> miller_rabin(561)
False
>>> miller_rabin(993905641)
False
```

Indications :

— la fonction `randint(x,y)` du module `random` permet de générer un entier aléatoire de l'intervalle $[\![x, y]\!]$.

— on n'utilisera pas l'algorithme de factorisation pour obtenir les entiers m et s (c'est trop long !) : faire des divisions successives par 2 de $N - 1$ suffit.

Remarque : Si après p essais on n'a pas trouvé de témoin, ceci signifie que N est premier avec une probabilité d'au moins $1 - \frac{\ln(m)}{4^p}$ (indépendamment de l'entier N choisi), qui converge très vite vers 1 lorsque p grandit. Ceci explique pourquoi 30 tests sont largement suffisants pour déclarer un entier de taille raisonnable premier. L'intérêt de ce test est qu'il est polynomial en la taille $\log(N)$ de N.

Exercice 10. Générer un entier premier (avec forte probabilité) de 1024 bits. On prendra par exemple des entiers impairs au hasard dans $[\![2^{1023} + 1, 2^{1024} - 1]\!]$, et on pourra vérifier s'ils sont premiers simplement avec le test PGP. On pourra aussi vérifier avec le test de Miller Rabin [4].

3 Cryptosystème RSA

Le cryptosystème RSA fait partie de la famille des cryptosystèmes à clé publique, dont le principe est le suivant : Alice veut envoyer un message M à Bob, qui possède une clé publique P_B (accessible à tous, elle se trouve par exemple dans un annuaire) et une clé qu'il garde secrète S_B. Ces deux clés sont des fonctions, vérifiant $S_B(P_B(M)) = M$ pour tout message M, et on suppose qu'il est difficile [5] de calculer S_B à partir de P_B. Ainsi, Alice va chiffrer son message à l'aide de P_B pour obtenir $M' = P_B(M)$. Si Charles (un attaquant) intercepte le message codé M', il ne pourra *a priori* pas trouver M facilement : seul Bob peut calculer $M = S_B(M')$.

La sécurité du cryptosystème RSA repose sur la difficulté du problème de la factorisation d'un entier, en particulier d'un produit de deux facteurs premiers : on suppose qu'il est impossible de calculer la factorisation d'un entier $N = pq$ où p et q sont deux entiers premiers dans un délai raisonnable, si p et q sont assez grands. Voici comment Bob peut générer son couple de fonctions (P_B, S_B) :

3. Un nombre d'un million de bits déclaré premier le sera effectivement avec une probabilité supérieure à 99.9999999999%.

4. Il vous faudra de l'ordre de 350 essais si vous ne prenez que des nombres impairs, avant de trouver un nombre premier. Pour générer un entier impair de 1024 bits, il suffit de générer d'abord un entier aléatoire de 1023 bits, le multiplier par 2 et ajouter 1 !

5. impossible dans un délai raisonnable.

— Il choisit deux grands entiers premiers p et q, ce qu'il est capable de faire s'il a fait la section précédente [6] ;

— Il calcule $N = pq$, et d un entier premier avec $N' = (p-1)(q-1)$;

— Il calcule l'inverse e de d modulo N', on a donc $de \equiv 1[N']$;

— Le couple (N, d) forme sa clé publique : en supposant que les messages sont des entiers de $[\![0, N-1]\!]$, P_B est la fonction :

$$\begin{array}{rccc} P_B : & [\![0, N-1]\!] & \longrightarrow & [\![0, N-1]\!] \\ & M & \longmapsto & M^d \bmod N \end{array}$$

La fonction S_B est alors donnée par

$$\begin{array}{rccc} S_B : & [\![0, N-1]\!] & \longrightarrow & [\![0, N-1]\!] \\ & M & \longmapsto & M^e \bmod N \end{array}$$

On peut montrer sans trop de difficulté que S_B et P_B sont inverses l'une de l'autre, car $M^{de} \equiv M[N]$, pour tout $M \in [\![0, N-1]\!]$. A priori, il est difficile de calculer e en connaissant seulement N et d. Pour encoder un message, il faut d'abord le découper en blocs d'entiers inférieurs à N (on l'écrit dans la base N, en fait) et coder séparémment chacun des blocs.

La fonction suivante (disponible sur le site web) permet de retrouver un message textuel à partir d'une liste d'entiers de $[\![0, N-1]\!]$:

```
def nombres_a_message(L,N):
    p=0
    for i in range(len(L)):
        p+=L[i]*N**i
    T=[]
    while p!=0:
        T.append(p%256)
        p//=256
    return "".join([chr(x) for x in T])
```

La liste suivante (également téléchargeable sur le site web) correspond à une liste de nombres où chacun a été encodé avec la clé publique $(N, d) = (960\,711\,889, 11\,604\,823)$:

```
[418683742, 760258171, 779124502, 460829648, 279753527, 22762789, 370394878, 726528724, 71499922,
565570180, 295296297, 870023005, 354545294, 746445650, 691735534, 413014530, 391643071, 183582893,
275476096, 379806142, 810232029, 236839939, 721906243, 614021459, 376223146, 902935861, 912609312,
170512961, 467806769, 100700137, 476615068, 277583213, 515739343, 938100374, 267239082, 253470644,
590021596, 570181568, 215023121, 960087269, 622681195, 621151163, 694795316, 546487635, 226610023,
839311923, 1]
```

Appliquer la fonction `nombres_a_message` à cette liste avec l'entier N renvoie une suite de caractères abscons : le message est codé ! Le nombre choisi N étant ridiculement petit, il est facile de « casser » le cryptosystème : la factorisation de N permet de retrouver p et q, et donc N'. L'entier e est un peu plus dur à déterminer : la relation $de \equiv 1[N']$, implique qu'il existe un entier k tel que $de + kN' = 1$. Les entiers e et k peuvent donc être calculés via l'algorithme d'Euclide *étendu* (détaillé ci-après). Une fois e déterminé, il est facile de retrouver le message originel (attention à utiliser la fonction d'exponentiation rapide modulo N écrite précédemment).

L'algorithme d'Euclide étendu. Cet algorithme est une généralisation de l'algorithme d'Euclide pour le calcul du PGCD de deux entiers a_0 et b_0, permettant en plus de calculer un couple de Bézout, c'est-à-dire deux entiers u et v tels que $\text{PGCD}(a_0, b_0) = ua_0 + vb_0$. L'idée est de procéder comme l'algorithme d'Euclide : on part de $a = a_0$ et $b = b_0$, et on effectue des divisions euclidiennes ($(a, b) \leftarrow (b, r)$, où r est le reste de la division euclidienne de a par b) jusqu'à ce que b soit nul. On utilise également 4 variables u, v, w et z qui satisfont les deux *invariants de boucle* suivants : $ua_0 + vb_0 = a$ et $wa_0 + zb_0 = b$. Lorsque b est nul, $\text{PGCD}(a_0, b_0) = a$ et u et v conviennent.

Exercice 11. Implémenter l'approche précédente, pour découvrir le message caché.

6. Les clés RSA d'aujourd'hui font par exemple 4096 bits, c'est-à-dire le produit de deux entiers premiers de 2048 bits.