MT10/P13 - TP2 : Autour de RSA

Table des matières

1 Manipulation de grands entiers

1. **Les plus grands entiers manipulables par `Sage`**
   Consulter le tutoriel pour voir comment mesurer le temps d'exécution d'une commande.
   Quel est le nombre de chiffre d'un entier de la forme $10^{(10^k)}$, où $k = 1, 2, \ldots$?
   Faire successivement :

   ```
   time a=10^10
   time a=10^(10^2)
   time a=10^(10^3)
    ...
   ```

   En déduire une évaluation du nombre de chiffres des plus grands entiers manipulables (explicitement) par `Sage` en temps raisonnable (disons moins de 5 secondes).
2. Chercher comment `Sage` retourne le nombre de chiffres (en base 10 ou autre) d'un entier.
3. Chercher comment `Sage` retourne $n!$ (factoriel de n) où n est un entier.

4. Donner le nombre de chiffres des nombres suivants (éviter d'afficher ces nombres encombrants à l'écran !) :

$$12^{34}, \quad (12^{34})^3, \quad 12^{(34^3)}, \quad 100!, \quad 1000!, \quad 10000!, \quad 100000!, \quad 6!!, \quad 7!!, \quad 10!!$$

5. `Sage` ne permet pas de calculer 100!!, on ne peut donc pas procéder comme précédemment pour connaître son nombre de chiffres. Utiliser la formule de Stirling :

$$\ln(n!) = n\ln n - n + \frac{1}{2}\ln n + \ln\sqrt{2\pi} + \frac{1}{12n} + O(\frac{1}{n^3})$$

pour écrire une fonction `Sage` qui retourne le nombre de chiffres de $n!$ d'un entier n passé en argument.

6. Calculer le nombre de chiffres du nombre de chiffres de 100!!. Evaluer le nombre d'écrans nécessaires pour afficher 100!! et le nombre de Go nécessaires pour stocker 100!!.

2 Bézout et Euclide

Consulter l'aide de `gcd` et de `xgcd`. Tester avec des nombres de Fibonacci (faire `fibonacci?`), vérifier que le nombre de divisions est alors maximal (voir exercice de TD correspondant). Tester aussi la taille des cœfficients de Bézout.

3 Nombres premiers

3.1 `Sage` et les nombres premiers

Les nombres premiers sont les notes de la musique des nombres. `Sage` possède quelques commandes très utiles pour travailler avec les nombres premiers : `is_prime`, `prime_range`, `next_prime`, `factor`, `prime_pi`. Consulter l'aide de ces commandes. Donner quelques exemples.

3.2 Nombres de Fermat

Les nombres de Fermat sont de la forme

$$F_n = 2^{(2^n)} + 1$$

Fermat [1] avait conjecturé que tous les F_n sont premiers. Montrer qu'il avait tort.

3.3 Nombres de Mersenne

Les nombres de Mersenne sont de la forme

$$M_p = 2^p - 1 \qquad p \text{ nombre premier}$$

1. **Pierre de FERMAT** (1601-1665). On lui doit de nombreux résultats en arithmétique et en probabilité, ainsi que quelques fameuses affirmations qui ont fait couler beaucoup d'encre. Si Euler montra que $F_5 = 4294967297 = 641 * 6700417$ réfutant ainsi la conjecture de Fermat annoncée dans le texte, il fallut attendre 1994 pour que Andrew Wiles établisse la preuve du "grand théorème de Fermat", à savoir : Pour tout entier $n \geq 3$, l'équation $x^n + y^n = z^n$ n'admet aucune solution (x, y, z) dans $(\mathbb{N}^*)^3$.

Marin Mersenne [2] affirma (en 1644, apprécier la performance !) que M_p est premier pour $p = 2, 3, 5, 7, 13, 17, 19, 31, 67, 127$ et 257, et composé pour les autres valeurs de p inférieures à 257. Euler écrit en 1732 que M_{41} et M_{47} sont premiers, mais il se trompe ...
L'objectif est de tester l'affirmation de Mersenne.

1. Former la liste des nombres premiers inférieurs ou égaux à 257. Combien y en a-t-il ? Former la liste des nombres de Mersenne correspondants.
2. Parmi la liste précédente, donner les nombres de Mersenne qui sont des nombres premiers. Rectifier l'affirmation de Mersenne.
3. Décomposer M_{41} et M_{47} en facteurs premiers.

3.4 Sur la répartition des nombres premiers

On sait, semble-t-il depuis Euclide (environ -300), que l'ensemble des nombres premiers est infini. On ne connait toujours pas de formule permettant de générer les nombres premiers, ni même seulement des nombres premiers arbitrairement grands. Leur répartition parmi les entiers reste mystérieuse [3] mais on dispose quand même du **théorème des nombres premiers (Hadamard et De La Vallée Poussin, 1896)** :

$$\lim_{n\to+\infty} \pi(n)\frac{\ln n}{n} = 1$$

où $\pi(n)$ désigne le nombre de nombres premiers $\leq n$.
Autrement dit, la densité des nombres premiers $\frac{\pi(n)}{n}$ tend à se comporter comme $\frac{1}{\ln n}$: c'est le phénomène de raréfaction des nombres premiers.

L'objectif est d'observer le comportement de suite $u_n = \pi(n)\frac{\ln n}{n}$, pour des n les plus grands possibles.

1. Evaluer les limites de la commande `prime_pi`. Pour cela, prendre successivement $n = 10^2, 10^3, \ldots$ et s'arrêter dès que le temps d'exécution de la commande devient de l'ordre de la minute.
2. Tracer sur un même graphe $n \mapsto \pi(n)$ et $n \mapsto \frac{n}{\ln n}$.
3. Tracer sur un graphe u_n en fonction de n. Commentaires ?

4 Algorithmes d'exponentiation

Il s'agit de programmer quelques algorithmes d'exponentiation et de comparer leur complexité empirique. Dans chaque cas, on comparera aussi avec la commande `Sage` usuelle `x^n` .

2. **Marin MERSENNE** (1588-1648). Moine de l'ordre des Minimes, grand esprit de son temps, a traduit de nombreux ouvrages de mathématiciens grecs.

3. On sait que pour tout entier $n \geq 1$, il existe un nombre premier p tel que $n < p \leq 2n$ (propriété appelée postulat de Bertrand, démontrée pour la première fois par Chebyshev en 1850). On sait aussi qu'il existe des trous arbitrairement grands entre deux nombres premiers consécutifs : pour tout entier $n > 1$, aucun de ces $n-1$ entiers consécutifs n'est premier : $n!+2$, $n!+3$, ..., $n!+n$. La recherche actuelle sur les nombres premiers est très active et laisse entrevoir des propriétés merveilleuses, liées à l'hypothèse de Riemann (les zéros non triviaux de la fonction zéta de Riemann

$$\zeta(s) = \sum_{n\geq 1} \frac{1}{n^s} = \prod_{p\ premiers} \frac{1}{1-p^{-s}}$$

sont tous sur la droite $Re(s) = \frac{1}{2}$).

4.1 Naïf itératif et naïf récursif

Ecrire une fonction `Sage` qui calcule x^n par multiplications successives par x. Cette fonction prendra comme arguments d'entrée x et n, et retournera x^n.
Ecrire un programme itératif, un autre récursif. Comparer leur complexité.
Remarque : Dans le modèle à coûts fixes, ces algorithmes ont une complexité en $O(n)$. Dans le modèle à coûts variables, ils ont une complexité en $O\left(n^2(\ln x)^2\right)$. (voir TD ?)

4.2 Dichotomique itératif et dichotomique récursif

On remarque d'abord que

$$x^n = \left| \begin{array}{ll} (x^2)^{\frac{n}{2}} & \text{si } n \text{ pair} \\ x(x^2)^{\frac{n-1}{2}} & \text{si } n \text{ impair} \end{array} \right.$$

ce qui suggère la règle de réécriture

$$(x, n, y) \longrightarrow \left| \begin{array}{ll} (x^2, \frac{n}{2}, y) & \text{si } n \text{ pair} \\ (x^2, \frac{n-1}{2}, y \cdot x) & \text{si } n \text{ impair} \end{array} \right.$$

qu'on initialise à $(x, n, 1)$ et qu'on applique tant que $n > 0$.
Ecrire un programme itératif, un autre récursif. Comparer leur complexité.
Remarque : Dans le modèle à coûts fixes, ces algorithmes ont une complexité en $O(\ln n)$, ce qui représente un gain considérable par rapport à l'algorithme naïf précédent. Cependant, dans le modèle à coûts variables, ces algorithmes ont tous la même complexité, en $O\left(n^2(\ln x)^2\right)$. D'où l'intérêt du calcul modulaire qui va suivre.

4.3 Algorithme d'exponentiation modulaire

Adapter l'algorithme précédent pour calculer $x^n \ \ [N]$. On rappelle que calculer modulo N revient à calculer avec des restes de divisions par N et que cela pourra se faire par la commande `mod`.
Ecrire une fonction `Sage` qui calcule $x^n \ \ [N]$. Elle devra prendre comme argument d'entrée l'entier N (en sus de x et n évidemment).
Comparer avec `power_mod`, ou une combinaison de `mod` et de `power` (ou `^`).
Remarque : Dans le modèle à coûts variables, l'exponentiation modulo N a une complexité en $O\left(\ln n(\ln N)^2\right)$. (voir TD ?)

4.4 Calcul des nombres de Fibonacci par exponentiation dans les matrices 2*2

On a vu en TD que

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$

Ecrire une fonction `Sage` qui calcule F_n par cette méthode. Pour ce faire, utiliser les facilités de `Sage` concernant les matrices et en particulier l'exponentiation.
Evaluer sa complexité empirique. `Sage` utilise-t-il un algorithme dichotomique pour calculer A^n quand A est une matrice ? Comparer aussi avec `fibonacci`.

Par curiosité, taper aussi les commandes suivantes :

```
power??
power_mod??
```

(retour décevant sur ma machine ...)

5 Cryptosystème RSA

5.1 Introduction

Rivest, Shamir et Adelman ont proposé dans [2] une méthode de cryptographie à clé publique dont on rappelle les grandes lignes :

(i) choix de deux (grands) nombres premiers p et q, qu'on gardera secrets ;

(ii) pour clé publique, former $N = pq$ et choisir e premier avec $\phi(N) = (p-1)(q-1)$. Le couple (N, e) sera rendu public et permettra le cryptage des messages ;

(iii) pour clé privée, on calcule un entier d tel que $ed = 1 \ [\phi(N)]$ par l'algorithme d'Euclide étendu.

On suppose que les messages sont codés en des listes d'éléments de $\mathbb{Z}/N\mathbb{Z}$. La fonction de cryptage (ou de chiffrement) est alors

$$\begin{array}{rcl} RSA_{(N,e)} : \mathbb{Z}/N\mathbb{Z} & \longrightarrow & \mathbb{Z}/N\mathbb{Z} \\ x & \longmapsto & x^e \end{array}$$

tandis que le déchiffrement se fera par $RSA_{(N,d)}$.

Ainsi chacun peut chiffrer (puisque (N, e) est publique) mais seul celui qui connait d peut déchiffrer. En effet, dans l'état actuel des connaissances, le calcul de d (à partir de N et e) nécessite celui de $\phi(N)$, donc de factoriser N, ce qu'on ne sait pas faire en temps raisonnable. D'où l'efficacité de la méthode RSA (jusqu'à présent ?).

L'objectif du TP est de construire un système de cryptographie RSA en utilisant les facilités offertes par `Sage`.

5.2 L'ensemble des messages, codage, décodage

Comme les fonctions de chiffrement-déchiffrement RSA travaillent sur $\mathbb{Z}/N\mathbb{Z}$, l'ensemble des messages admissibles est constitué des suites d'entiers compris entre 0 et $N-1$ (pour avoir le représentant canonique des classes d'entiers modulo N). Bien sûr, ces suites d'entiers devront coder pour des messages initiaux écrits en français.

`Sage` propose trois monoïdes (libres) de chaînes de caractères (revoir TD) dans (1) l'alphabet usuel (26 lettres), (2) l'alphabet hexadécimal (0 à 9, puis a,b,c,d,e,f) et (3) l'alphabet binaire (0 et 1). Essayons.

```
S1=AlphabeticStrings();
S2=HexadecimalStrings();
S3=BinaryStrings();
```

Par exemple, la séquence de commandes,

```
original="Que j'aime à faire connaître un nombre utile aux sages ...";
ori_num=S3.encoding(original);
print ori_num;
print ori_num.decoding();
```

retourne à l'affichage :

```
0101000101110101011001010010000001101010001001110110000101101001011011\
01
0110010100100000110000111010000000100000011001100110000101101001011100\
10
0110010100100000011000110110111101101110011011100110000111000011101011\
10
0111010001110010011001010010000001110101011011100010000001101110011011\
11
0110110101100010011100100110010100100000011101010111010001101001011011\
00
0110010100100000011000010111010101111000001000000111001101100001011001\
11
0110010101110011001000000010111000101110001011\
10
Que j'aime à faire connaître un nombre utile aux sages ...
```

Ce codage est injectif (de même que $S2$, mais pas $S1$: essayez vous-même !), et on va l'utiliser. Il faudra ensuite découper cette longue chaîne binaire en paquets susceptibles d'être transformés en entiers compris entre 0 et $N-1$. Cette liste d'entiers, tous compris entre 0 et $N-1$, constituera le message à crypter.

Ecrire une fonction `numerise` qui réalise cette transformation. Elle admettra comme paramètres d'entrée :

- le message original, *i.e.*une chaîne de caractères usuels ;
- le modulo N choisi (auquel il faudra adapter la taille des paquets !).

En sortie, elle produira une liste d'entiers, tous compris entre 0 et $N-1$.

Ecrire une fonction `alphabetise` qui réalise l'opération inverse.

5.3 La génération de clés RSA

C'est un point crucial du dispositif RSA : il faut choisir des couples de nombres premiers (p, q) tels qu'il soit impossible de calculer p et q à partir de la connaissance de leur produit $n = pq$.

Evidemment, il faut choisir p et q assez grands pour que leur produit n ne puisse pas être factorisé par la fonction `factor` de `Sage`. Evaluer les limites de `factor`.
[Indication : attention, ce n'est pas si facile. En particulier, il faut s'affranchir des nombres à petits facteurs qui se factorisent rapidement, même s'ils ont plus de 100 chiffres décimaux.]

A l'aide d'une recherche Wiki, Google, ...
Quelles sont les recommandations de la société RSA Data-Security ? Pourquoi les nombres de Mersenne premiers doivent être évités ? Pourquoi l'exposant de chiffrement e ne doit pas être choisi trop petit ?

Ecrire une fonction `cleRSA` qui délivre un ensemble de paramètres RSA (N, e, d) qui tiennent compte des recommandations précédentes. Elle prendra en entrée un entier m qui sera un minorant du nombre de chiffre du N délivré. Produire quelques clés avec N de 30 chiffres, et indiquer le temps mis par `factor` pour casser ces clés. Votre fonction `cleRSA` peut-elle générer des clés RSA de 2048 bits (réputées incassables) ?

5.4 Fonctions de chiffrement et déchiffrement RSA

Elles sont de même nature (une exponentiation modulaire) et s'écrivent en une seule ligne dans `Sage`. Elles transforment une liste d'entiers compris entre 0 et $N-1$ en une liste de même nature.

[Indication : utiliser `map` et `power_mod`.]

5.5 Signature avec RSA

On va utiliser deux protocoles de signature. Pour les présenter, considérons deux individus Alice et Bob, de paramètres respectifs (N_A, e_A, d_A) et (N_B, e_B, d_B), qui désirent s'échanger des messages. On suppose Nc fixé et que $3Nc$ (= le nombre de chiffres nécessaire pour coder un bout de message) vérifie $3Nc < N_A$ et $3Nc < N_B$. Alice envoie un message à Bob. Ce message ne doit pas être lu par un tiers, Bob doit savoir le lire et il doit être sûr qu'Alice est bien à l'origine du message.

1. **Protocole 1 :** message+signature
   (i) Alice numérise son texte (m1) et sa signature (s1). Elle obtient (m1c,s1c).
   (ii) Alice chiffre m1c avec la clé publique de Bob (N_B, e_B) et s1c avec sa clé privée (N_A, d_A). Elle obtient (m2c,s2c) qu'elle transmet à Bob.
   (iii) Bob décrypte m2c avec sa clé privée (N_B, d_B) (il est le seul à pouvoir le faire) et décrypte s2c avec la clé publique d'Alice (N_A, e_A) (il est sûr que le message vient d'Alice car elle seule a pu ainsi chiffrer sa signature). Il retrouve ainsi (m1c,s1c).
   (iv) Bob alphabétise m1c et s1c pour retrouver le texte d'Alice m1 et sa signature s1.

2. **Protocole 2 :** message signé
   Alice applique un double cryptage à son message, l'un avec la clé publique de Bob (seul Bob pourra décrypter le message), l'autre avec sa clé privée (seule Alice peut le faire). Mais attention : pour que Bob puisse retrouver le message original, ces deux opérations doivent se faire dans le bon ordre.

   Si $N_A > N_B$:
   (i) Alice numérise son texte m1, elle obtient m1c.
   (ii) Alice crypte m1c avec la clé publique de Bob (N_B, e_B), elle obtient m2c, qu'elle chiffre ensuite avec sa clé privée (N_A, d_A) pour obtenir m3c qu'elle transmet à Bob.
   (iii) Bob applique d'abord la clé publique d'Alice pour récupérer m2c, auquel il applique sa clé privée pour récupérer m1c.
   (iv) Enfin, Bob alphabétise m1c et retrouve le texte d'Alice m1.

   Si $N_A < N_B$:
   (i) Alice numérise son texte m1, elle obtient m1c.
   (ii) Alice crypte m1c d'abord avec sa clé privée (N_A, d_A) et obtient m2c. Elle chiffre ensuite m2c avec la clé publique de Bob (N_B, e_B) et obtient m3c qu'elle transmet à Bob.
   (iii) Bob applique d'abord sa clé privée pour récupérer m2c, puis la clé publique d'Alice pour récupérer m1c.
   (iv) Enfin, Bob alphabétise m1c et retrouve le texte d'Alice m1.

Ecrire des fonctions `protocole1` et `protocole2` qui réalisent la signature (ou la lecture selon les paramètres RSA fournis en entrée) en mettant en oeuvre les deux protocoles ci-dessus.

Expliquer pourquoi il faut considérer deux cas dans le protocole 2. Pourquoi n'a-t-on pas considéré le cas $N_A = N_B$?

Références

[1] Jean-Paul Delahaye, *Merveilleux nombres premiers*, Belin, Pour la science, 2000.

[2] R. Rivest, A. Shamir, L. Adleman, *A Method for Obtaining Digital Signatures and Public Key Cryptosystems*, Comm. ACM **21**, p. 120-126, 1978.