Programmation applicative – L2
Deuxième série de TP : Un système cryptographique probabiliste (Blum-Goldwasser)

G.Artignan {artignan@lirmm.fr} M.Lafourcade {lafourcade@lirmm.fr}
S.Daudé {sylvain.daude@univ-montp2.fr} A.Chateau {chateau@lirmm.fr}
B.Paiva Lima da Silva {bplsilva@gmail.com} C.Dony {dony@lirmm.fr}

1 Préambule

Cet enoncé concerne 3 séances de TP. L'évaluation des TPs se fera en deux phases :

- Pendant les TPs, votre présence et votre progression seront notées. Il y a 10 séances de TP, la note de présence est donc sur 10. Seules les absences justifiées seront acceptées. À la fin de chaque séance de TP, vous indiquerez sur la feuille de présence le numéro de la dernière question à laquelle vous avez entièrement répondu.
- Pendant la dernière séance, une évaluation de votre travail tout au long des séances sera effectuée, sous la forme d'un petit problème auquel il faudra répondre en vous servant des fonctions que vous aurez écrites durant les TPs. Nous noterons cette évaluation finale également sur 10.

En conséquence, il faudra sauvegarder précieusement votre travail au fur et à mesure des séances, pour pouvoir l'utiliser lors de l'examen final.

2 Principe du cryptosystème à clef publique

Nous allons mettre en place un système cryptographique, qui permet de s'échanger des messages codés sans que les personnes qui les interceptent accidentellement ou volontairement puisse les décoder en un temps raisonnable. Les systèmes cryptographiques sont utilisés depuis des millénaires, notamment par les militaires.

À l'heure actuelle, on utilise la puissance de calcul des ordinateurs pour "craquer" les cryptosystèmes qui ne sont pas assez sûrs ou utilisent des clefs de dimension trop faibles. Plusieurs méthodes cryptographiques existent, certains d'entre vous connaissent peut-être le système RSA [1] qui est l'un des plus connu. Le système que nous allons mettre en place durant ce TP est un "système à clef publique" (comme RSA) car il repose sur les principes suivants :

- Chaque personne voulant utiliser ce système se fabrique deux "clefs", l'une publique qu'elle diffuse à tous ses interlocuteurs, l'autre privée qui lui servira à décoder les messages qu'on lui envoie.
- Si Alice veut envoyer un message m à Bernard, elle utilise la clef publique de Bernard comme paramètre d'un *algorithme d'encryption* appliqué au message m. Elle obtient un **message codé** c, qu'elle envoie à Bernard.
- Si Bernard veut décoder le message c envoyé par Alice, il utilise un *algorithme de décryption* avec comme paramètre *sa* clef privée à lui, pour retrouver m à partir de c.
- Si un indélicat Christian cherche à savoir ce qu'Alice dit à Bernard, il ne faut pas que la connaissance de la clef publique de Bernard et du message codé c permette de retrouver facilement m. C'est pourquoi la décryption doit être un problème **difficile** du point de vue du temps de calcul, sauf dans le cas où on connaît des éléments supplémentaires (la clef privé) qui permette de racourcir le temps de calcul.

1. du nom de ses trois inventeurs, Ron *R*ivest, Adi *S*hamir et Leonard *A*dleman

Le cryptosystème de Blum-Goldwasser [2] repose sur la difficulté qu'il y a, étant donné un nombre $n = pq$ très grand produit de deux premiers, à retrouver ses facteurs premiers p et q.

De plus, le cryptosystème de Blum-Goldwasser est un cryptosystème *probabiliste*, c'est-à-dire que certains choix se feront au hasard, ce qui rend encore plus difficile la décryption.

Nous travaillerons sur trois aspects du cryptosystème, et nous utiliserons abondamment la récursivité et la structure de données de listes :

1. La génération des clefs (à la fin de cette étape vous pourrez diffuser votre clef publique et conserverez soigneusement votre clef privée)
2. L'algorithme d'encryption, où comment transformer un texte en français en une liste d'éléments codés
3. L'algorithme de décryption, qui permet de réaliser l'opération inverse bien évidemment.

2.1 Génération de clefs

Voici l'algorithme général de génération de clef pour le cryptosystème de Blum-Goldwasser :

1. Prendre au hasard deux grand nombres premiers p et q, tous les deux égaux à 3 modulo 4
2. Calculer $n = pq$
3. Utiliser l'algorithme d'Euclide étendu pour calculer deux entiers u et v tels que $up + vq = 1$
4. La clef publique est n, la clef privée est (p, q, u, v)

Nous allons détailler ci-dessous chaque étape de cet algorithme.

2.1.1 Algorithme d'Euclide étendu

L'algorithme d'Euclide [3] permet, étant donnés deux entiers p et q quelconques, de calculer leur pgcd (plus grand commun diviseur) d.

L'algorithme d'Euclide étendu permet, en plus du pgcd d, de calculer des entiers u et v tels que $up + vq = d$ (identité de Bézout). Il repose sur une relation de récurrence qui est la suivante :

- si $x' = 0$, $eucl(x, u, v, x', u', v') = (x, u, v)$
- sinon, $eucl(x, u, v, x', u', v') = eucl(x', u', v', x - (x \div x') * x', u - (x \div x') * u', v - (x \div x') * v')$

Exercice 1 *En transcrivant simplement la définition donnée ci-dessus, écrire la fonction* `eucl` *à 6 paramètres. On rappelle que les fonctions* `quotient` *et* `remainder` *sont fort utiles lorsque l'on parle de division euclidienne.*

Exemple : `(eucl 13 1 0 19 0 1)` doit renvoyer `(1 3 -2)`.

Ensuite, on peut montrer qu'une telle fonction `eucl` permet de calculer le PGCD de deux nombres ainsi que les deux coefficients de Bézout associé. En effet, on a l'égalité suivante :

$$eucl(p, 1, 0, q, 0, 1) = (pgcd(p, q), u, v) \qquad \text{avec } u \text{ et } v \text{ tels que } pgcd(p, q) = p * u + q * v.$$

Exercice 2 *On vous demande maintenant d'écrire la fonction* `euclide_etendu` *à deux paramètres p et q qui renvoie la liste (p, q, u, v). Dans la suite du TP, une telle liste constituera une clef privée.*

Exemple : `(eucl 13 1 0 19 0 1)` doit renvoyer `(1 3 -2)` et `(euclide_etendu 13 19)` doit renvoyer `(13 19 3 -2)`, car $3 \times 13 + (-2) \times 19 = 1$.

2. Le cryptosystème de Blum-Goldwasser (BG) est un algorithme de chiffrement proposé par Manuel Blum et Shafi Goldwasser en 1984

3. Euclide, en grec ancien Eukleidês (né vers -325, mort vers -265 à Alexandrie) est un mathématicien de la Grèce antique, auteur des *Éléments*, qui sont considérés comme l'un des textes fondateurs des mathématiques modernes.

2.1.2 Trouver des nombres premiers

Nous nous intéressons aux nombres premiers qui sont congrus à 3 modulo 4. Dans un premier temps, nous allons générer tous les nombres premiers inférieurs à un entier donné N, puis nous filtrerons ceux qui nous intéressent, avant d'en sélectionner deux au hasard.

La méthode de génération de nombres premiers choisie pour ce TP est le crible d'Ératosthène [4], méthode brutale mais efficace... on considère tous les entiers entre 2 et N. On enlève tous ceux qui sont divisibles par 2 (sauf 2 qui est premier), puis on enlève tous ceux qui sont divisibles par 3, sauf 3, et ainsi de suite, jusqu'à enlever tous ceux qui sont divisibles par N.

Exercice 3 *On peut s'arrêter à $\sqrt{N}$. Pourquoi ?*

Exemple, voici le crible d'Ératosthène pour $N = 25$:

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25  l
2 3   5   7   9    11    13    15    17    19    21    23    25  l1 (crible 2 l)
2 3   5   7        11    13          17    19          23    25  l2 (crible 3 l1)
2 3   5   7        11    13          17    19          23        l3 (crible 5 l2)
```

Exercice 4 *Écrire une fonction* `genere_liste` *prenant un paramètre N, qui renvoie la liste des entiers entre 2 et N (bornes comprises). Si le paramètre est ≤ 1, la fonction doit renvoyer la liste vide.*

Exercice 5 *Écrire une fonction* `crible` *prenant deux paramètres, une liste l et un entier d, qui renvoie la liste l dans laquelle on a enlevé tous les éléments qui sont divisibles par d. C'est une fonction typique de traitement des listes, elle fonctionne sur un schéma classique utilisant un* `cond` *:*

- *Si la liste est vide, on renvoie la liste vide (Rappel, le test pour savoir si une liste l est vide est* `(null? l)`*)*
- *Si le premier élément de la liste* `(car l)` *est divisible par d, on renvoie l'appel récursif de la fonction sur la fin de la liste* `(cdr l)` *(on a enlevé le premier élément)*
- *Sinon, on "garde" l'élément en renvoyant la concaténation de* `(car l)` *et de l'appel récursif de la fonction sur la fin de la liste* `(cdr l)`

Exercice 6 *Écrire la fonction* `eratosthene` *qui prend en paramètre un entier N et renvoie la liste des entiers premiers inférieurs ou égaux à N. On aura sans doute besoin de définir une fonction auxiliaire qui travaille récursivement sur une liste.*

Exemple, voici le résultat de `(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)
```

Il faut maintenant filtrer notre liste d'entiers premiers, pour ne garder que ceux qui sont congrus à 3 modulo 4.

Exercice 7 *Sur le même principe que la fonction* `crible`*, écrire une fonction* `filtre_3modulo4` *qui prend en paramètre une liste et renvoie la liste contenant les mêmes éléments sauf ceux dont le modulo 4 est différent de 3.*

Exemple, voici le résultat de `(filtre_3modulo4 (eratosthene 100))`

```
(3 7 11 19 23 31 43 47 59 67 71 79 83)
```

Pour sélectionner deux nombres au hasard dans la liste des entiers premiers congrus à 3 modulo 4, il faut pouvoir sélectionner un élément de rang donné dans une liste.

4. Ératosthène (en grec ancien Eratosthénês) était un astronome, géographe, philosophe et mathématicien grec du IIIe siècle av. J.-C.

Exercice 8 *Écrire une fonction* `extraction` *qui prend une liste l et un entier i en entrée, et renvoie l'élément de rang i de la liste, ou bien 0 si i n'est pas dans les limites de la liste.*

Exercice 9 *Écrire une fonction* `cherche_pq` *qui prend en paramètre un entier N, et donne une liste de deux entiers premiers p et q, choisis au hasard,* **distincts***, congrus à 3 modulo 4, et inférieurs ou égaux à N. On peut utiliser un* `let*` *pour simplifier les manipulations et éviter de calculer plusieurs fois le crible d'Ératosthène et le filtrage. Il faudra utiliser également la fonction* `random` *(lire l'aide de cette fonction pour s'en servir à bon escient).*

Attention : deux appels successifs à la fonction `random` pourront tout à fait retourner le même nombre. Il vout faut prévoir ce cas et, lorsqu'il se produit, reboucler sur un ou des appel(s) à `random`.

Exercice 10 *Écrire une fonction* `genere_cles` *qui prend en paramètre un entier N et qui renvoie une liste (n, p, q, u, v), où $n = pq$, et le reste constituant une clef privée valide pour la problème de Blum-Goldwasser. Choisissez une clef telle que la clef publique est à 8 chiffres (pour $N = 10000$ on peut en trouver), gardez votre clef privée dans un endroit caché (par exemple un fichier texte dans votre répertoire de travail, sinon vous allez oublier...), et diffusez votre clef publique (n) à vos camarades.*

2.2 Algorithme d'encryption

Pour l'instant nous supposerons que les messages sont donnés sous la forme d'une liste de 0 et de 1, nous verrons plus tard comment encoder les caractères alphabétiques en suites de 0 et de 1.

On s'attèle à la création une fonction qui va encoder une suite de caractères en un message codé, en utilisant la clef publique du destinataire. On part du principe que la clef publique est un entier s'écrivant sur 8 chiffres en base 10. On détaille ci-dessous le processus qui va mener du message en clair, m, au message codé c.

Si Bernard veut envoyer un message $m = m_1 m_2 \dots m_t$ (suite de 0 et de 1) à Alice dont la clef publique est n_A, voici comment il procède :

1. On prend un entier r au hasard entre 1 et $n_A - 1$ et on prend $x_0 = r^2 \bmod n_A$. **Le nombre $\mathbf{x_0}$ s'appelle un résidu quadratique modulo $\mathbf{n_A}$.**
2. Calculer une suite de nombres $x_1, \dots, x_t$ tels que $x_i = x_{i-1}^2 \bmod n_A$
3. Calculer une suite de 0 ou 1 : $p_1, \dots, p_t$ tels que p_i vaut 0 si x_i est pair, 1 si x_i est impair
4. Pour chaque m_i chiffre binaire du message, calculer $c_i = p_i$ XOR m_i
5. Calculer $x_{t+1} = x_t^2 \bmod n_A$
6. Envoyer le message codé $c = (c_1, c_2, \dots, c_t, x_{t+1})$ à Alice.

Exercice 11 *Écrire la fonction* `graine` *qui prend un paramètre entier n_A et calcule un résidu quadratique modulo n_A au hasard.*

Exercice 12 *Écrire la fonction* `suite_residus` *qui prend en paramètre un entier x_0, un entier n_A et un entier t et renvoie une liste $(x_1, \dots, x_{t+1})$ calculée comme dans l'algorithme ci-dessus.*

Exemple : pour $x_0 = 4721616$, $n_A = 33439193$ et $t = 8$, on doit obtenir la liste :

```
(15191900 26639265 6947837 21234085 11828808 13376788 18622257 15023475 12569981)
```

Exercice 13 *Écrire la fonction* `encrypte` *qui prend en paramètre la liste m représentant le message, ainsi que l'entier n_A, et renvoie le message encodé $c = (c_1, c_2, \dots, c_t, x_{t+1})$. Petite indication : l'opérateur binaire "XOR" est implémenté par la fonction* `bitwise-xor`.

2.3 Algorithme de décryption

Voici ce qu'il faut faire pour décrypter un message c crypté avec la clef publique n_A, connaissant p et q tels que $n_A = pq$:

1. Calculer $d_1 = ((p+1)/4)^{t+1} \bmod (p-1)$ et $d_2 = ((q+1)/4)^{t+1} \bmod (q-1)$
2. Calculer $a = x_{t+1}^{d_1} \bmod p$ et $b = x_{t+1}^{d_2} \bmod q$
3. Calculer $x_0 = ubp + vaq \bmod n_A$
4. Calculer une suite de nombres $x_1, \ldots, x_t$ tels que $x_i = x_{i-1}^2 \bmod n_A$
5. Calculer une suite de 0 ou 1 $p_1, \ldots, p_t$ tels que p_i vaut 0 si x_i est pair, 1 si x_1 est impair
6. Pour chaque c_i chiffre binaire du message crypté, calculer $m_i = p_i$ XOR c_i

Remarquez qu'après l'obtention de la graine x_0, tout se passe de manière presque semblable à l'encryption...

Exercice 14 *Écrire une fonction* `graine_decryption` *qui prend en paramètre la clef privée* (p, q, u, v)*, et le message encrypté* c*, et renvoie le* x_0 *donné dans l'algorithme.*

Exercice 15 *Écrire la fonction* `decrypte` *qui prend en paramètre la liste* c *représentant le message codé, ainsi que la clef privée* (p, q, u, v)*, et renvoie le message décodé* $m = (m_1, m_2, \ldots, m_t)$*.*

Exercice 16 *Testez que votre algorithme fonctionne bien en encryptant et décryptant une liste de 0 et de 1.*

Remarquez que si vous encryptez plusieurs fois de suite la même suite de 0 et de 1, vous n'obtenez pas le même message codé (à cause du choix aléatoire de la graine). Cela permet d'éviter des attaques dites "à texte choisi", dans lesquelles on essaie de deviner la façon dont le texte est codé en testant plusieurs textes particuliers.

A Optionnel : passage d'un texte français à une liste de 0 et de 1

Pour le moment, le cryptosystème est assez peu ludique dans le sens où on ne peut pas s'échanger des petits mots cryptés entre camarades, à moins de parler couramment le binaire. Nous allons, dans cette partie, nous doter d'outils de conversion entre les caractères et les listes de 0 et de 1. Les caractères en scheme sont codés en unicode, dont nous n'utilisons en français qu'une faible partie. À chaque caractère correspond un entier (son code en unicode), et comme nous allons encrypter des listes de 0 et de 1, il faut transformer ces entiers correspondant au codage des caractères en suites de 0 et de 1. Pour nous simplifier la tâche, nous considérons un codage binaire sur deux octets (i.e. 16 bits) des entiers, ce qui nous autorise à utiliser des entiers entre 0 et $2^{16} - 1 = 65535$.

Pour rentrer une chaîne de caractère dans une variable, on utilise la fonction `read`. Par exemple :

```
(define texte (read))
```

donne la main à l'utilisateur, et rentre dans la variable `texte` le texte entré au clavier jusqu'à ce qu'on tape la touche entrée.

Inversement, pour écrire un texte stocké dans une variable, on connaît déjà la fonction `display`.

Voici quelques autres fonctions qui nous seront fort utiles (regardez donc dans l'aide de DrScheme comment l'on s'en sert...) :

```
string->list          string->list
char->integer         integer->char
```

Exercice 17 *Écrire une fonction* `string->integer-list` *qui transforme une chaîne de caractères en une liste d'entiers correspondant aux codage unicode des caractères de la chaîne (on utilisera la fonction* `map`*), et une fonction* `integer-list->string` *qui réalise l'opération inverse.*

Une fois qu'on sait convertir une chaîne de caractères en entier, il reste à convertir chacun de ces entiers en une liste de 0 et de 1.

Pour cela, on utilise un algorithme de changement de base dit "par divisions successives".

Exemple :

diviseur	*reste*
58	0
29	1
14	0
7	1
3	1
1	1
0	

$\rightarrow 111010$

$$\begin{aligned} 58 &= 29 \times 2^1 + 0 \times 2^0 \\ 58 &= 14 \times 2^2 + 1 \times 2^1 \\ 58 &= 7 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 \\ 58 &= 3 \times 2^4 + 1 \times 2^3 + 1 \times 2^1 \\ 58 &= 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \end{aligned}$$

On réalise une succession de divisions par 2, puis on lit la représentation binaire du nombre en considérant les restes de bas en haut.

Exercice 18 *Écrire une fonction* `integer->digit-list` *qui transforme un entier en la suite de 0 et de 1 représentant son codage binaire sur deux octets, quitte à rajouter des 0 à gauche. Pensez à tester si l'entier à convertir est bien dans les limites de l'encodage (*$[0, \ldots, 65535]$*).*

Exercice 19 *Écrire une fonction* `string->digit-list` *qui transforme une chaîne de caractères en une liste de 0 et de 1 constituée de la concaténation de toutes les listes correspondant aux codages binaires sur deux octets des entiers obtenus à partir des caractères de la chaîne de caractères.*

Pour l'opération inverse, on effectue l'addition des puissances de la base, ici 2 : Exemple :

$$101101 = 1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3 + 0 \times 2^4 + 1 \times 2^5 = 1 + 4 + 8 + 32 = 45.$$

Pour calculer ce polynôme, on peut utiliser une méthode un peu plus intelligente que de calculer à chaque étape la puissance de 2 qui correspond au coefficient, à savoir la méthode de Horner [5].

Soit le polynôme $a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$. La méthode de Horner consiste à écrire le polynôme sous la forme :

$$((\ldots((a_n x + a_{n-1})x + a_{n-2})x + \ldots)x + a1)x + a_0$$

Cette méthode permet de ne pas stocker les coefficients du polynôme en mémoire, les coefficients étant lus dans l'ordre décroissant des indices et utilisés immédiatement dans le calcul. C'est exactement ce qu'il nous faut, car on peut "lire" facilement les coefficients qui correspondent au codage binaire d'un entier (on utilise le `car` de la liste des 16 chiffres 0 ou 1 qui correspondent à cet entier, puis on relance l'algorithme sur la suite de la liste). La méthode consiste donc à multiplier le premier coefficient par x et à lui ajouter le second coefficient. On multiplie alors le nombre obtenu par x et on lui ajoute le troisième coefficient, etc.

Ainsi, si on connaît la valeur du calcul res_k au rang k et la liste des coefficients qu'il reste à utiliser, le calcul au rang $k+1$ est $res_{k+1} = b \times res_k +$ le premier coefficient de la liste, et on relance le calcul sur la fin de la liste.

Exercice 20 *Écrire une fonction* `horner` *qui prend en entrée une liste de coefficients et un entier b et qui calcule la valeur du polynôme correspondant aux coefficients en ce point b.*

Exercice 21 *Écrire une fonction* `decoupe` *qui prend une liste de 0 et de 1 et renvoie une liste de listes de 16 chiffres 0 ou 1 (et qui teste si ce découpage peut se faire !). On peut se servir de la fonction* `list-tail` *(voir dans l'aide).*

Exercice 22 *Écrire une fonction* `digit-list->integer` *qui transforme une suite de 0 et de 1 en un entier dont c'est la représentation binaire.*

Exercice 23 *Écrire une fonction* `digit-list->string` *transformant une liste de 0 et de 1 en une chaîne de caractères.*

Exercice 24 *Testez vos fonctions... et utilisez votre cryptosystème pour communiquer avec vos camarades.*

Références

[Eucl] Sur l'algorithme d'Euclide étendu (calcul des coefficients de Bezout) : `http://fr.wikipedia.org/wiki/Algorithme_d'Euclide_Étendu`

[BG] Une petite introduction au cryptosystème de Blum et Goldwasser : `http://fr.wikipedia.org/wiki/Cryptosystème_de_Blum-Goldwasser`

[BGcomplete] Explication du cryptosystème, et preuve de l'encodage et du décodage (en anglais) :

1. M. Blum, S. Goldwasser, "An Efficient Probabilistic Public Key Encryption Scheme which Hides All Partial Information", Proceedings of Advances in Cryptology - CRYPTO '84, pp. 289-299, Springer Verlag, 1985.

2. Menezes, Alfred ; van Oorschot, Paul C. ; and Vanstone, Scott A. Handbook of Applied Cryptography. CRC Press, October 1996. ISBN 0-8493-8523-7

Le chapitre 8 contenant l'explication du cryptosystème de Blum et Goldwasser est en téléchargement libre : `http://www.cacr.math.uwaterloo.ca/hac/about/chap8.pdf`

5. William George Horner (1786 - 22 septembre 1837) est un mathématicien britannique. Il est né à Bristol en Angleterre et est mort à Bath en Angleterre.

Horner est connu pour "sa" méthode (déjà publiée par Zhu Shijie vers 1300, mais aussi utilisée (en Angleterre !) par Isaac Newton 150 ans avant Horner) qui permet d'évaluer rapidement un polynôme en un point et pour son invention en 1834 du zootrope, un appareil optique donnant l'illusion du mouvement.