Algorithme RSA
avec Maple

Par Sadik BOUJAIDA

Aspect théorique

RSA est une méthode de cryptographie à clés asymetrique, une clé publique qui permet de crypter un message et une autre privée qui permet de déchiffrer le message cryptée. La clé publique est connue de tout le monde (divulguée dans un annuaire par exemple) ce qui fait que tout un chaqu'un peut l'utiliser pour crypter un message. Alors que la clé privée est en principe détenue par la seule personne qui devrait étre capable de déchiffrer le message.

Dans la pratique la méthode RSA, puisque couteuse en calcul, est surtout utilisée pour créer un canal sécurisée entre deux correspondant, en permettant par exemple de crypter et d'échanger la clé d'un système à clés symetrique.

La force de l'algorithme sur lequel repose RSA tient d'une propriété des nombres premiers. Il est facile de déterminer si un nombre est premier ou pas, mais trés difficile de connaitre ses facteurs premiers dans le cas où il est composé. Etant donné donc un couple de nombres premier très grand p et q, il serait très difficile à quelqu'un qui connait leurs produit $n = pq$ de retrouver p et q.

On calcule donc des nombres premiers p et q très grand, puis un nombre e qui est premier avec $a = (p-1)(q-1)$. et ensuite un entier u tel que $ue = 1 \ \mathbf{mod}\ a$ (ce qui n'est possible que lorsque e est premier avec a). Le couple (n, e) est alors la clé public alors que $(n, u)\, b$ représente la clé privée. Remarquer que puisque il est difficile de calculer p et q à partir de n, il est très difficile de calculer a et donc aussi u, même connaissant e.

L'algorithme repose alors sur un résultat arithmétique (le théorème d'Euler) qui implique que pour tout entier x premier avec n, $x^a = 1 \ \mathbf{mod}\ n$.

Etant donnée un message m, on calcule son cryptogramme $x = m^e \ \mathbf{mod}\ n$ à l'aide de la clé publique. Pour inverser le message crypté x, il suffit alors de faire $x^u = m^{eu} = m^1 = m \ \mathbf{mod}\ n$. On retrouve ainsi le message original m. En fait on retrouve le résidu de m modulo n, d'où la nécessité de diviser le message en plusieurs parties s'il dépasse n.

Choix des clés :

On calcule des nombres premiers p et q, en utilisant la procédure Maple *nextprime* , qui permet de calculer le nombre premier qui suit immediatement l'argument qu'on lui passe.

il suffit alors de générer des nombres aléatoire d'assez grande taille (80 chiffre dans ce TP) en utilisant la procédure *rand* et de les passer en arguments à *nextprime*. Les nombres–messages ne devrait pas dépasser alors 160 digits (la taille de n).

```
> restart:
> nextprime(rand(10^80)());nextprime(rand(10^80)());
```

$$99596883909465520335617712166467577059926068276305009675299763431340064249105\backslash 379$$
$$34633129088861013688643347733731974176548310874222673528866121995231476726634\backslash 233$$

On définit ensuite p et q en dur en effectuant un copier/coller, ainsi il ne seront pas changé par une nouvelle exécution de la feuille de calcul.

```
> p:=
  18726572777229977543680250315429551323873309942840276296876328434054665953292
  163; q:=
  95776487115595252869654675633582239004006819141942972603144709740085577863190
  299;
```

$$p := 18726572777229977543680250315429551323873309942840276296876328434054665953\backslash 292163$$
$$q := 95776487115595252869654675633582239004006819141942972603144709740085577863\backslash 190299$$

On peut même les protéger de toute modification avec l'instruction :

```
> protect('p','q');      # on peut ensuite les "d"protèger avec unprotect('p',
  'q')
```

Si on essaie de les modifier :

```
> p:=0;
Error, attempting to assign to `p` which is protected
> n_key:=p*q; a_key:=(p-1)*(q-1);
```

$$n\_key := 17935653563176237557884741169191512944973411052124618296908272182999247287\backslash 04909560372545045727013552087110176846988319176438819721189853213901390000\backslash 433214326737$$
$$a\_key := 17935653563176237557884741169191512944973411052124618296908272182999247287\backslash 04909445869485152901783138752184227835197991296309734937940953192863215860\backslash 189397844276$$

```
> protect('n_key','a_key');
```

Un choix simple de e premier avec a, consistera en le choix de e lui même premier, il serait alors certain qu'il est premier avec a. On pourrait pousser la recherche de simplicité en prenant e comme un nombre de Fermat ie de la forme $2^k + 1$ (il parait qu'il y'en a beaucoup qui sont premier), son écriture binaire aura ainsi un maximum de zéro.

```
> k:=10:e:=2^k+1 : while not isprime(e) do k:=k+1; e:=2^k+1 od;
```

$$k := 11$$
$$e := 2049$$
$$k := 12$$
$$e := 4097$$
$$k := 13$$
$$e := 8193$$
$$k := 14$$

$$e := 16385$$
$$k := 15$$
$$e := 32769$$
$$k := 16$$
$$e := 65537$$

```
> e_key:=e;
```

$$e\_key := 65537$$

```
> isprime(e_key);
```

$$true$$

```
> convert(e_key,binary);
```

$$10000000000000001$$

Pour le calcul de u on exploite le potentiel de Maple avec l'instruction :

```
> u_key:=1/e_key mod a_key;
```

$$u\_key := 125497850900494793123261756838978776739497644173105058400799950790514744776570536428638754683029539639835378695757624375352664166881545219593708118621092233310 9$$

```
> igcd(u_key,e_key);
```

$$1$$

Cryptage et décryptage d'un nombre :

Signalons d'abords un problème . Puisque on va calculer des nombres de la forme m^u avec a qui est de taille conséquente, on risque de dépasser la capacité mémoire à la disposition de Maple si on fait naivement $m\hat{}u$ **mod** n. On peut définir nous même une procédure qui calcule les puissances de m modulo n de façon itérative, mais Maple offre une alternative rapide. Le problème dans l'exécution directe de l'instruction $m\hat{}u$ **mod** n; c'est que Maple va user des regles de simplifcation automatique pour d'abords calculer m^u avant d'analyser complétement l'instruction, il suffit alors d'inhiber cette simplification en faisant précéder l'opérateur ^ d'un &. (utilisé notament dans l'évaluation d'un produit matriciel).

```
> m:=rand(10^159)();
```

$$m := 821549589959220446689526267208268647719020161417040321534896243841673191981376063404698722790093420089439314027256726392672837555201348290193577189177729852590$$

```
> m^u_key mod n_key; # qu'est ce qu'on disait
Error, numeric exception: overflow
> t:=time(): m&^u_key mod n_key; time()-t; # rapide comme l'éclair.
```

$$8428862475376935371243462195907534193024028675557006800909032754211752896448723327601745281477321923819659671221847704942779734372192513260845793138313675577916$$

$$0.004$$

On écrit maintenant la procédure *EncDec* qui s'occupera de l'encodage et du décodage selon les arguments qu'on lui passera.

```
> EncDec:=proc(x::nonnegint,y::posint,z::posint)::posint;
    x&^y mod z;
  end proc;
```

$$EncDec := \mathbf{proc}(x::nonnegint, y::posint, z::posint)::posint;\quad mod(x\,\&\hat{}\,y, z)\quad \mathbf{end\ proc}$$

```
> x:=EncDec(m,e_key,n_key);
```

$x :=$
5953983847199534410456475687007399930473502973807755116791476395509275764\
0\
6380521727784272572559789936780346093421579174363164142168191871749933826\
4\
1116341281

On vérifie maintenant si le décodage permet de retrouver le message original :

```
> evalb(EncDec(x,u_key,n_key)=m);
```

$$true$$

Cryptage d'une chaine de charactères :

Il s'agit maintenant de transformer un message texte en un nombre, et de faire subir à ce nombre la procédure d'encodage. Pour cela on remplacera chaque charactère du message par son code hexadécimal de la table ASCII utilisée. On concatène les codes hexadécimaux et reconvertit le nombre ainsi obtenu en décimal. Une table ASCII étant codé en 8 bits (7 en fait) chaque charactère devrait être representé par un code à un ou deux chiffres hexadécimaux. Pour ceux qui sont representé par un seul chiffre on ajoute un 0 à droite (non signifcatif), chaque charactère devant être representée par exactement deux symboles hexadécimaux pour éviter tout décalage lors de la procédure de décodage.

Pour résoudre la contrainte de la taille maximale des nombres obtenus qui ne doit pas dépasser *n_key*, on pourrait par exemple, avant de traduire le message en un nombre, le subdiviser en plusieurs portions de longueur maximale un certain entier *t*. De telle façon qu'une fois que chaque portion est convertie en un nombre, celui-ci soit plus petit que *n_key*. Pour cela il suffirait que $t < \frac{1}{2}\ length(\ convert(\ n\_key, hex)\ )$. (la moitié de la longueur hexadécimale de *n_key*, puisque chaque charactère est representé par deux chiffres héxadécimaux).

```
> length(convert(floor(n_key),hex));
```

$$\frac{133}{2}$$

On pourrait par exemple prendre t=64.

On écrit maintenant la procédure (récursive) Subd qui éclate une chaine en une séquence de sous chaines de longueur maximale *t*:

```
> Subd:=proc(CH,t::posint)
  if length(CH)<=t then RETURN(CH)
  else substring(CH,1..t),Subd(substring(CH,t+1..length(CH)),t)
  fi
  end proc  ;
```

Subd := **proc**(*CH*, *t*::*posint*)
  if *length*(*CH*) <= *t* **then**
    RETURN(*CH*)
  else
    substring(*CH*, 1 ..*t*), *Subd*(*substring*(*CH*, *t* + 1 ..*length*(*CH*)), *t*)
  end if
end proc

On teste la procédure *Subd*

```
> Subd("abcdefghijklmnopqrstuvwxyz",6);
```

"abcdef", "ghijkl", "mnopqr", "stuvwx", "yz"

Un message qui va servir d'exemple par la suite.

```
> Message:="Ceci est un message pour le test, avec & et $.\n";
```

Message := "Ceci est un message pour le test, avec & et $.
"

On subdivise.

```
> Mess:=Subd(Message,64); # pas besoin de le faire ici, le message est trop
  court.
```

Mess := "Ceci est un message pour le test, avec & et $.
"

On traduit chaque portion en une liste de nombrex décimaux representant chacun le code ASCII d'un charactère.

```
> L_dec:=convert(Mess,bytes);
```

L_dec := [67, 101, 99, 105, 32, 101, 115, 116, 32, 117, 110, 32, 109, 101, 115, 115, 97, 103, 101, 32, 112, 111, 117, 114, 32, 108, 101, 32, 116, 101, 115, 116, 44, 32, 97, 118, 101, 99, 32, 38, 32, 101, 116, 32, 36, 46, 10]

On convertit en hexadécimal

```
> L_hex:=map(convert,L_dec,hex); #Noter le charactère de fin de ligne \n qui
  est codé par un seul chiffre A=13,
```

L_hex := [*43*, *65*, *63*, *69*, *20*, *65*, *73*, *74*, *20*, *75*, *6E*, *20*, *6D*, *65*, *73*, *73*, *61*, *67*, *65*, *20*, *70*, *6F*, *75*, *72*, *20*, *6C*, *65*, *20*, *74*, *65*, *73*, *74*, *2C*, *20*, *61*, *76*, *65*, *63*, *20*, *26*, *20*, *65*, *74*, *20*, *24*, *2E*, *A*]

On écrit maintenant la procédure *RightZero* qui va ajouter un 0 non significatif à chaque code hexadécimal qui comporte un seul chiffre.

```
> RightZero:=proc(H)
  if length(H)=1 then cat(0,H) else H fi
  end proc;
```

RightZero := **proc**(*H*) **if** *length*(*H*) = 1 **then** *cat*(0, *H*) **else** *H* **end if** **end proc**

```
> RightZero(1);RightZero(A);
```

01
0A

et on applique à *L_hex*.

```
> L_hex:=map(RightZero,L_hex);
```

L_hex := [*43*, *65*, *63*, *69*, *20*, *65*, *73*, *74*, *20*, *75*, *6E*, *20*, *6D*, *65*, *73*, *73*, *61*, *67*, *65*, *20*, *70*, *6F*, *75*, *72*, *20*, *6C*, *65*, *20*, *74*, *65*, *73*, *74*, *2C*, *20*, *61*, *76*, *65*, *63*, *20*, *26*, *20*, *65*, *74*, *20*, *24*, *2E*, *0A*]

L_hex est une liste qui contient maintenant les codes hexadécimaux bien formatés des charactères du message.

On concatène on revient à une écriture decimale:

```
> N_hex:=cat(op(L_hex));
```

N_hex := *436563692065737420756E206D6573736167652070F7572206C6520746573742C2061766*
5632026206574202420E0A

```
> Mess_nbr:=convert(N_hex,decimal,hex);
```

Mess_nbr := *405203171733550077803994294005229211810745157692299908743190392921539948 44*
99090779085013176194828560030254944 6154

On encode maintenant :

```
> Mess_nbr_code:=map(EncDec,Mess_nbr,e_key,n_key);
```

Mess_nbr_code :=
808132011446621835153196520736718577787412578932648048882132955197270335835\
656803200716716846958233445136208722596978014240179263042997238045663896745\
44662016329

Il faut maintenant retraduire ce nombres en une chaine de charactères pour obtenir le message codé sous forme de texte. Pour cela on Mess_nbr_code en hexadécimal, on l'eclate en une listes de paires de charactères, on traduit chaque paire en décimal et on convertit tout en une chaine de charactère avec *convert* / *bytes*. Tout ceci dans la procédure NbrToString.

Noter que puisque on regroupe les charactère par paires, on devrait s'assurer que la conversion en hexadécimal produit un nombre avec une longueur qui est paire. Pour cela on ajoute un 0 non significatif au début si cette condition n'est pas remplie.

```
> NbrToString:=proc(N::posint)
  local N_hex,L_hex,L_dec;
  #option trace;
  N_hex:=convert(N,hex);
  if length(N_hex) mod 2 = 1 then N_hex:=cat(0,N_hex) fi;
  L_hex:=[seq(substring(N_hex,2*k-1..2*k),k=1..length(N_hex)/2)];
  L_dec:=map(convert,L_hex,decimal,hex);
  convert(L_dec,bytes)
  end proc;
```

NbrToString := **proc**(*N*::*posint*)
local *N_hex*, *L_hex*, *L_dec*;
N_hex := *convert*(*N*, *hex*);
if *mod*(*length*(*N_hex*), 2) = 1 **then** *N_hex* := *cat*(0, *N_hex*) **end if**;
L_hex := [*seq*(*substring*(*N_hex*, 2 * *k* − 1 ..2 * *k*), *k* = 1 ..1 / 2 * *length*(*N_hex*))];
L_dec := *map*(*convert*, *L_hex*, *decimal*, *hex*);
convert(*L_dec*, *bytes*)
end proc

```
> Mess_code:=map(NbrToString,Mess_nbr_code);
```

Mess_code := "□q=-=□□s□□□T□□3Q□□o□□-□□□□□2□□□□□t□□□Ew□l□□A□□□□□□t#□□□□YI"

Exercice :

- Ecrire la procédure StringToNbr qui prend en argument une chaine de charactère M et produit un nombre representant cette chaine.
- Ecrire la procédure EncMess qui prend en argument un message en chaine de charactères et retourne le message crypté (en chaine de charactères).

Solution

```
> StringToNbr:=proc(M)
  local L_dec,L_hex,N_hex;
  L_dec:=convert(M,bytes);
  L_hex:=map(RightZero@convert,L_dec,hex);
  N_hex:=cat(op(L_hex));
  convert(N_hex,decimal,hex)
  end proc;
```

StringToNbr := **proc**(*M*)

```
    local L_dec, L_hex, N_hex;
    L_dec := convert(M, bytes);
    L_hex := map(`@`(RightZero, convert), L_dec, hex);
    N_hex := cat(op(L_hex));
    convert(N_hex, decimal, hex)
end proc
```

```
> EncMess:=proc(M::string)
  local S_messd,L_messc,L_dec,L_dec_code;
  S_messd:=Subd(M,64);
  L_dec:=map(StringToNbr,[S_messd]);
  L_dec_code:=map(EncDec,L_dec,e_key,n_key);
  L_messc:=map(NbrToString,L_dec_code);
  cat(op(L_messc))
  end proc;
```

```
EncMess := proc(M::string)
    local S_messd, L_messc, L_dec, L_dec_code;
    S_messd := Subd(M, 16);
    L_dec := map(StringToNbr, [S_messd]);
    L_dec_code := map(EncDec, L_dec, e_key, n_key);
    L_messc := map(NbrToString, L_dec_code);
    cat(op(L_messc))
end proc
```

Décryptage du message codé

La procédure pour le décryptage est similaire à celle utilisée pour le cryptage, à ceci près, qu'on doit utiliser *EncDec* pour décoder et non pour coder . Tout cela dans la procédure DecMess.

```
> DecMess:=proc(MC::string)
  local S_messc,L_messd,L_dec,L_dec_decode;
  S_messc:=Subd(M,64);
  L_dec:=map(StringToNbr,[S_messc]);
  L_dec_decode:=map(EncDec,L_dec,u_key,n_key);
  L_messd:=map(NbrToString,L_dec_decode);
  cat(op(L_messd))
  end proc;
```

```
DecMess := proc(MC::string)
    local S_messc, L_messd, L_dec, L_dec_decode;
    S_messc := Subd(M, 64);
    L_dec := map(StringToNbr, [S_messc]);
    L_dec_decode := map(EncDec, L_dec, u_key, n_key);
    L_messd := map(NbrToString, L_dec_decode);
    cat(op(L_messd))
end proc
```

N.B : *Après vérification sur un exemple, la procédure DecMess ne donne pas le résultat escompté, à savoir le message en clair d'origine. Ceux qui sont interessés pourront télécharger le fichier en version Maple RSA_Maple_ws.mw ou en version PDF RSA_Maple_ws.pdf qui reprend juste les programmes de cette feuille revus et corrigés et augmenté d'une partie qui traite des entrées sorties (lire un fichier texte, l'encoder et l'enregistrer dans un deuxième fichier texte, et inversement) sur le site de mon collègue My Smail ELMAMOUNI en cliquant directement sur les liens dans ce document, ou contacter l'auteur par email à l'adresse* ***jsadikster@gmail.com.***