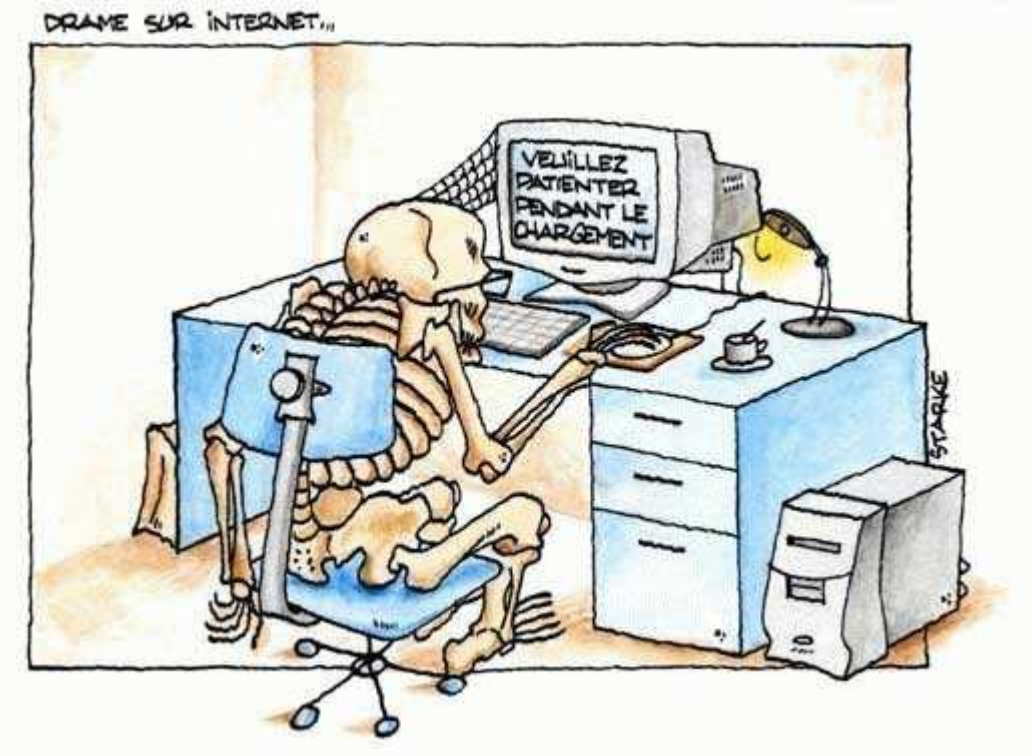
DESTREE Lucile
MARCHAL Mickaël
P2 Groupe B

Mini-RSA

Programme d'initiation au chiffrement RSA



Projet de Mathématiques pour l'Informatique N°1

Sommaire

Introduction

Le cryptage est historiquement l'une des premières applications de l'informatique. Ce domaine, qui était il y a encore quelques années, réservé aux militaires et aux grandes entreprises, concerne aujourd'hui tous ceux qui souhaitent transmettre des données protégées, qu'ils soient professionnels ou particuliers. Pour cela, il existe de nombreuses méthodes de cryptage, mais peu d'entre elles sont reconnues comme sûres. La méthode RSA fait depuis longtemps partie de cette catégorie.

Dans ce projet, nous allons développer un petit programme de cryptage et décryptage basé sur ce chiffre. Le but ne sera pas de développer un programme au code « incassable », mais plutôt de comprendre comment fonctionne le cryptage RSA.

Présentation du cryptage RSA

Le cryptage RSA, du nom de ses concepteurs, Ron Rivest, Adi Shamir et Leonard Adleman, est le premier algorithme de chiffrement asymétrique. Il a été découvert en 1977 au Massachusetts Institute of Technology.

Un chiffrement asymétrique est un cryptage où l'algorithme de chiffrement n'est pas le même que celui de déchiffrement, et où les clés utilisées sont différentes. L'intérêt est énorme : il n'y a plus besoin de transmettre la clé à son destinataire, il suffit de publier librement les clés de cryptage. N'importe qui peut alors crypter un message, mais seul son destinataire, qui possède la clé de décodage, pourra le lire. En quelques années, RSA s'est imposé pour le cryptage comme pour l'authentification et a progressivement supplanté son concurrent, le DES.

Le RSA est basé sur la théorie des nombres premiers, et sa robustesse tient du fait qu'il n'existe aucun algorithme de décomposition d'un nombre en facteurs premiers. Alors qu'il est facile de multiplier deux nombres premiers, il est très difficile de retrouver ces deux entiers si l'on en connaît le produit.

Principe

1) Génération des clés

Le RSA fonctionne à partir de deux nombres premiers, que l'on appellera **p** et **q**. Ces deux nombres doivent être très grands, car ils sont la clé de voûte de notre cryptage. Aujourd'hui, on utilise des clés de 128 à 1024 bits, ce qui représente des nombres décimaux allant de 38 à 308 chiffres !

Une fois ces deux nombres déterminés, multiplions-les. On note **n** le produit $\mathbf{n = p \times q}$, et $\mathbf{z = (p - 1) \times (q - 1)}$.

Cherchons maintenant un nombre **e** (inférieur à **z**) , qui doit nécessairement être premier avec **z**.

Calculons ensuite l'inverse de **e** modulo **z**, que nous noterons **d**.

$$\mathbf{d \equiv e^{-1} \bmod((p - 1)(q - 1))}$$

Le couple **(e, n)** est la clé publique, et **(d, n)** est la clé privée.

2) Cryptage et décryptage

Pour crypter un nombre, il suffit de le mettre à la puissance **e**. Le reste modulo **n** représente le nombre une fois crypté.

$$\mathbf{c = t^e \bmod n}$$

Pour décrypter, on utilise la même opération, mais en mettant à la puissance **d** :

$$\mathbf{t = c^d \bmod n}$$

Une fois **e**, **d** et **n** calculés, on peut détruire **p**, **q** et **z**, qui ne sont pas nécessaires pour crypter et décrypter. Pire encore, on peut calculer très rapidement la clé privée **d** à partir de **p** et **q**, il ne faut donc pas conserver ces nombres.

Note : En général, la clé privée est ensuite cryptée à l'aide d'un cryptage symétrique. Cela permet de la conserver de façon sûre, car la clé utilisée par le cryptage symétrique n'a pas à être transmise, et donc ne risque pas d'être interceptée.

3) Exemple

Voici un exemple de l'utilisation de RSA, avec des petits nombres :

Saddam souhaiterait envoyer le message suivant à George :
« Kisses from Iraq ». Malheureusement, Vladimir les espionne, et pourrait intercepter ce message. Nos deux compères vont donc crypter leurs échanges avec la méthode RSA.

George a choisi **p = 37** et **q = 43**. Il en déduit n = 37 × 43 = 1591, et z = 36 × 42 = 1512.

Il choisit ensuite **e = 19**, qui est premier avec 1512. L'inverse de 19 modulo 1512 est **d = 955**.

George peut donc maintenant publier ses clés publiques, par exemple sur son site internet : e = 19 et n = 1591

Saddam va utiliser ces clés pour crypter son message, mais il doit avant tout convertir son texte en une suite de nombres. Comme Saddam veut envoyer le message sous forme d'un fichier informatique, le mieux est d'utiliser le code ASCII (Amercian Standard Code for Information Interchange). Ce code attribue un nombre entre 0 et 255 pour chaque caractère de base utilisable sur un ordinateur.

En ASCII, « Kisses from Iraq » devient :

K	**i**	**s**	**s**	**e**	**s**		**f**	**r**	**o**	**m**		**l**	**r**	**a**	**q**
75	105	115	115	101	115	32	102	114	111	109	32	43	114	97	113

Il suffit à Saddam de coder chaque nombre comme expliqué ci-dessus. Il obtient :

75^{19} [1591] = 371 ; 105^{19} [1591] = 1338 ; etc…

75	**105**	**115**	**115**	**101**	**115**	**32**	**102**	**114**	**111**	**109**	**32**	**43**	**114**	**97**	**113**
371	1338	1410	1410	1174	1410	930	1397	632	703	483	930	1405	632	532	1441

Saddam envoie cette suite de nombres à George, qui va le décrypter avec sa clé **d**. Il va pouvoir retrouver le message original :

371^{955} [1591] = 75 ; 1338^{955} [1591] = 105 ; etc…

371	**1338**	**1410**	**1410**	**1174**	**1410**	**930**	**1397**	**632**	**703**	**483**	**930**	**1405**	**632**	**532**	**1441**
75	105	115	115	101	115	32	102	114	111	109	32	43	114	97	113
K	**i**	**s**	**s**	**e**	**s**		**f**	**r**	**o**	**m**		**l**	**r**	**a**	**q**

En recodant en ASCII, George va pouvoir lire le message de son ami, sans que Vladimir n'ait pu le déchiffrer.

4) Forcage du code RSA

Si Vladimir est vraiment curieux et veut absolument décrypter le message de Saddam, il devra essayer de forcer le code RSA. Pour cela, il faut qu'il détermine la clé privée **d** à partir de ce qu'il connaît, c'est à dire **n** et **e** (qui sont publiques).

Pour ce faire, il doit tenter de trouver **p** et **q** à partir de **n**. Comme nous le disions précédemment, il n'existe pas de méthode miraculeuse pour retrouver ces deux nombres. Il faut tenter toutes les combinaisons de nombres premiers pour trouver celle dont le produit donnera **n** . Selon le principe fondamental de la théorie des nombres, la décomposition en facteurs premiers est unique, Vladimir sera donc sûr d'avoir trouvé les bonnes valeurs de **p** et **q**.

Une fois **p** et **q** trouvé, il ne lui reste plus qu'a déterminer **z**, et de calculer **d** à partir de **e** et **z**, de la même façon que lors de la génération de la clé. Vladimir pourra alors décoder le message crypté et ainsi assouvir sa curiosité.

Bien sûr, pour p = 37 et q = 43, l'opération de forçage ne prendra que quelques millisecondes. Mais si Saddam et George avaient employé des clés de 1024 bits, le pauvre Vladimir aurait du patienter quelques mois avant de pouvoir déchiffrer le fameux message !

L'algorithme ne semble donc pas très compliqué. Sa démonstration fait appel à des théorèmes comme le théorème d'Euclide ou celui de Fermat.

Démonstration

1) Théorème de Fermat

Le petit théorème de Fermat affirme que si **p** est un nombre premier, alors pour tout entier **a,** on a :

$$a^p \equiv a \pmod{p}$$

ou $\mathbf{a^p \pmod p \equiv a}$

ou $\mathbf{a^{p-1} \pmod p \equiv 1}$

ou $\mathbf{a^{p-2} \pmod p \equiv a^{-1}}$

Ce qui signifie que si vous considérez un entier **a**, et que vous le multipliez par lui-même **p** fois et que vous lui soustrayez **a**, le résultat est divisible par **p**.

2) Généralisation du théorème par Euler

L'indicateur d'Euler, noté **Φ(n)**, est le nombre des nombres premiers avec **n** compris entre **1** et **n-1**.

$$\Phi(n) = \left(1 - \frac{1}{p1}\right) \cdot \left(1 - \frac{1}{p2}\right) \Lambda \left(1 - \frac{1}{pk}\right) si\ n = p1 \times p2 \times \Lambda \times pk$$

Chaque facteur premier de **n** n'est utilisé qu'une seule fois, quelque soit le nombre de fois qu'il apparaît dans la décomposition de **n**.

On a alors :

$\mathbf{a^{\Phi(n)} \pmod n \equiv 1}$

ou $\mathbf{a^{\Phi(n)-1} \pmod n \equiv a^{-1}}$

on a :

$\mathbf{a^{\Phi(n)-1} \times a = a^{\Phi(n)-1+1} = a^{\Phi(n)}}$

Donc $\mathbf{a^{\Phi(n)}}$ est un inverse de de **a (mod n)**

On a donc, grâce au théorème de Fermat-Euler :

$$
\begin{aligned}
c_i^{\,d} \pmod n &\equiv (t_i^{\,e})^{d} \pmod n \\
&\equiv t_i^{\,ed} \pmod n \\
&\equiv t_i^{\,k(p-1)(q-1)+1} \pmod n \\
&\equiv t_i * t_i^{\,k(p-1)(q-1)} \pmod n \\
&\equiv t_i * 1 \pmod n \\
&\equiv t_i \pmod n
\end{aligned}
$$

Puisque

$$t_i^{\,k(p-1)(q-1)} \pmod n \equiv 1$$

Les limites du RSA

Dans la pratique, le RSA est un code sûr, si l'on respecte les quelques règles suivantes :

- **p** et **q** doivent être très grands. En effet, on estime qu'il faut moins d'une seconde pour casser un code à base de nombres de 32 bits. Comme expliqué précédemment, on doit utiliser des clés de 128 bits minimum (ou plus si la législation en vigueur le permet), pour garantir un code sûr à court terme.

- Il faut crypter le message par blocs de plusieurs caractères. En effet, si l'on crypte caractère par caractère, chacun d'eux sera codé par le même nombre. Il est alors facile de forcer le message par analyse fréquentielle*, ou en testant systématiquement les 256 combinaisons possibles. En codant par groupe de 4 ou 8 caractères (32 ou 64 bits), l'analyse fréquentielle devient impossible, et le nombre de combinaisons possibles augmente énormément (plus de 4 milliards de combinaisons sur 32 bits).

Le RSA possède également quelques inconvénients d'ordre mathématique :

- On ne peut pas choisir un **n** inférieur à la valeur maximale à coder. Ainsi, si on code caractère par caractère, on code des valeurs comprises entre 0 et 255. Si n < 255, notre cryptage ne sera pas bijectif. Cela s'explique car nous effectuons le modulo **n** lors du cryptage et du décryptage. Si **n** est trop petit, plusieurs caractères pourront être cryptés par le même nombre, et ne pourront donc plus être différenciés lors du décryptage.

- Les calculs sont souvent très lourds, du fait de la taille des entiers à manipuler. Le cryptage d'un message long, avec des clés de grande taille, peut prendre plusieurs heures sur un ordinateur puissant.

De plus, lorsqu'il s'agit de programmer un logiciel de cryptage / décryptage RSA, nous nous heurtons à un nouveau problème : le dépassement de capacité des variables. En effet, le plus grand type d'entier prévu dans le langage C est le type **unsigned long int**, qui ne peut gérer que des nombres de 32 bits au maximum. Ceci est notoirement insuffisant pour un cryptage efficace.

Notre programme se contentera donc de fonctionner pour des « petits nombres », c'est à dire pour des nombres **p** et **q** compris entre 0 et 255. Nous allons donc obtenir une clé **n** comprise entre 0 et 65535. Mais pour les raisons évoquées ci-dessus, il ne sera pas possible de crypter par blocs de plusieurs caractères (il nous faudrait n > 65536 pour pouvoir coder par blocs de 2 caractères).

Notre code est donc d'une grande simplicité à casser, c'est pour cela que notre programme n'a qu'un but pédagogique et ne devra en aucun cas être utilisé pour un « usage commercial ».

** Note : l'analyse fréquentielle consiste à déchiffrer un message en se basant sur la fréquence d'apparition des lettres dans une langue. Ainsi, dans un texte en français, le code apparaissant le plus souvent dans un message chiffré sera vraisemblablement celui correspondant à la lettre « e ».*

Choix algorithmiques

Nous allons maintenant décrire les algorithmes utilisés dans le logiciel :

1) Génération des clés

a) Génération de p et q

Il existe quelques méthodes permettant de déterminer si un nombre est premier ou non. Ces méthodes ont une complexité et un taux de réussite variables (certaines fonctionnent à 100%, et d'autres ne donnent qu'une *probabilité* que le nombre soit premier).

Dans le cas de petits nombres, la méthode la plus efficace est sans nul doute l'application de la définition d'un nombre premier :

> ***Premier commandement :***
> ***" Un nombre est premier si et seulement si il ne possède aucun diviseur hormis 1 et lui même ."***

Pour un nombre donné, il suffit donc de calculer les restes de tous les entiers > 1 qui sont strictement inférieurs à sa racine carrée. Si l'un d'eux a un reste nul, il est diviseur, et le nombre n'est pas premier. Dans le cas contraire, le nombre sera bien évidemment premier.

Nous pouvons donc écrire l'algorithme suivant :

```
EstPremier(n : entier long ; premier : booléen)
$1
      booléen premier
      premier <- VRAI

      POUR i <- 2 JUSQU'A i < √n FAIRE
      $11
            SI reste de n modulo i = 0 ALORS
            $111
                  premier <- FAUX
                  INTERROMPRE
            111$
            i <- i+1
      11$
      RENVOYER premier
1$
```

Il nous suffit maintenant de choisir un nombre au hasard, et tester si il est premier ou non. S'il ne l'est pas, on choisit au hasard un nouveau nombre, et ainsi de suite jusqu'à en obtenir un premier. On répète cette opération deux fois, pour **p** et pour **q**.

```
FAIRE
$1
      FAIRE
      $11
            p <- ( 3+rand() ) mod PQ_MAX
      11$
      TANT QUE (estPremier(p) ≠ 1)

      FAIRE
      $12
            q <- ( 3+rand() ) mod PQ_MAX
      12$
      TANT QUE (estPremier(q) ≠ 1)
1$
TANT QUE (p = q) OU (p*q ≤ 256)
```

Deux tests sont effectués une fois p et q trouvés :

- On vérifie que $p \neq q$. En effet, si p = q, le code peut être plus facile à forcer.
- On vérifie que $p \times q > 256$, afin que notre programme soit bijectif (cf. *limites du RSA*).

Si l'une de ces deux conditions n'est pas remplie, on choisit deux autres nombres p et q.

De plus, bien que premiers, les nombres 1 et 2 ne conviennent pas pour p et q. En effet, si p ou q = 1, on a z = 0, et si p = 1 on a z = (q-1), ce qui est très facile à casser.

A partir de p et q, on calcule n, puis z.

Note : cette méthode possède un défaut : statistiquement, un nombre a plus de chances d'être premier si il est petit. Notre programme aura donc tendance à tirer au sort des petits nombres (du moins, petits par rapport à la limite maximum qu'on lui aura fixé). Pour remédier à cela, il aurait fallu calculer tous les nombres premiers compris entre 0 et cette limite, les stocker dans un tableau, et en choisir deux au hasard. Cette méthode est beaucoup plus longue et beaucoup plus lourde que celle que nous avons utilisé.

b) Génération de e

La clé publique e doit impérativement être première avec z. Nous devons donc construire un algorithme capable de déterminer si deux nombres sont premiers entre eux, c'est à dire que PGCD(a,b) = 1.
La méthode la plus simple et la plus rapide pour déterminer un PGCD reste sans conteste l'algorithme de ce cher Euclide, que nous pouvons programmer de la manière suivante :

```
Pgcd (entier long a, entier long b ; entier long pgcd)
$1
      entier long c
      c <- 0

      c <- reste de a modulo b
      SI c = 0 ALORS
      $11
            RENVOYER b  // a est multiple de b, le pgcd est donc b
      11$
      SINON
      $12
            TANT QUE c ≠ 0 FAIRE
            $121
                  a <- b
                  b <- c
                  c <- reste de a modulo b
            121$
            RENVOYER b
      11$
1$
```

Nous pouvons alors choisir un nombre e < z au hasard et déterminer si celui-ci convient(⇔ Pgcd(e,z) = 1). Si ce n'est pas le cas, on choisit un autre e jusqu'à en trouver un correspondant :

```
FAIRE
$1
      e <- (3 + rand()) % z
1$
TANT QUE(Pgcd(e, z) ≠ 1)
```

d) Génération de d

Le calcul de d, la clé privée, est l'opération la plus lourde. Rappelons tout d'abord ce qu'est l'inverse d'un nombre modulo z :

$$d \text{ inverse de } e \text{ modulo } z \Leftrightarrow d = e^{-1} [z] \Leftrightarrow \mathbf{d \times e = 1 [z]}$$

On utilise alors la boucle suivante pour déterminer d :

```
d <- 0
TANT QUE (reste de (e*d) modulo z) ≠ 1 FAIRE
$1
            d <- d+1
1$
```

On rappelle qu'il existe nécessairement un inverse pour e, car e est tel que e et z sont premiers.

Nous avons donc généré nos clés publiques et privées et pouvons donc nous attaquer au cryptage d'un message.

2) Cryptage

Notre programme est capable de coder n'importe quel fichier : texte, image, exécutable ou encore MP3 pirate. Nous n'allons donc pas pouvoir raisonner en termes de caractères, mais en termes d'octets. Notre programme va lire le fichier octet par octet, coder cet octet en un nouveau nombre sur deux octets (16 bits), puis l'écrire dans un nouveau fichier, qui sera notre fichier crypté. Il est nécessaire de coder sur 16 bits car chaque octet du fichier d'origine va devenir un nombre compris entre 0 et n (on a $n < 65535 = 2^{16} - 1$). La taille de notre fichier crypté sera donc multipliée par deux.

Le cryptage fonctionne avec la formule suivante :

$$c_i = t_i^{\,e}\ [n]$$

Caractère crypté sur 2 octets — Caractère à crypter (1 octet)

Cela correspond à l'algorithme suivant :

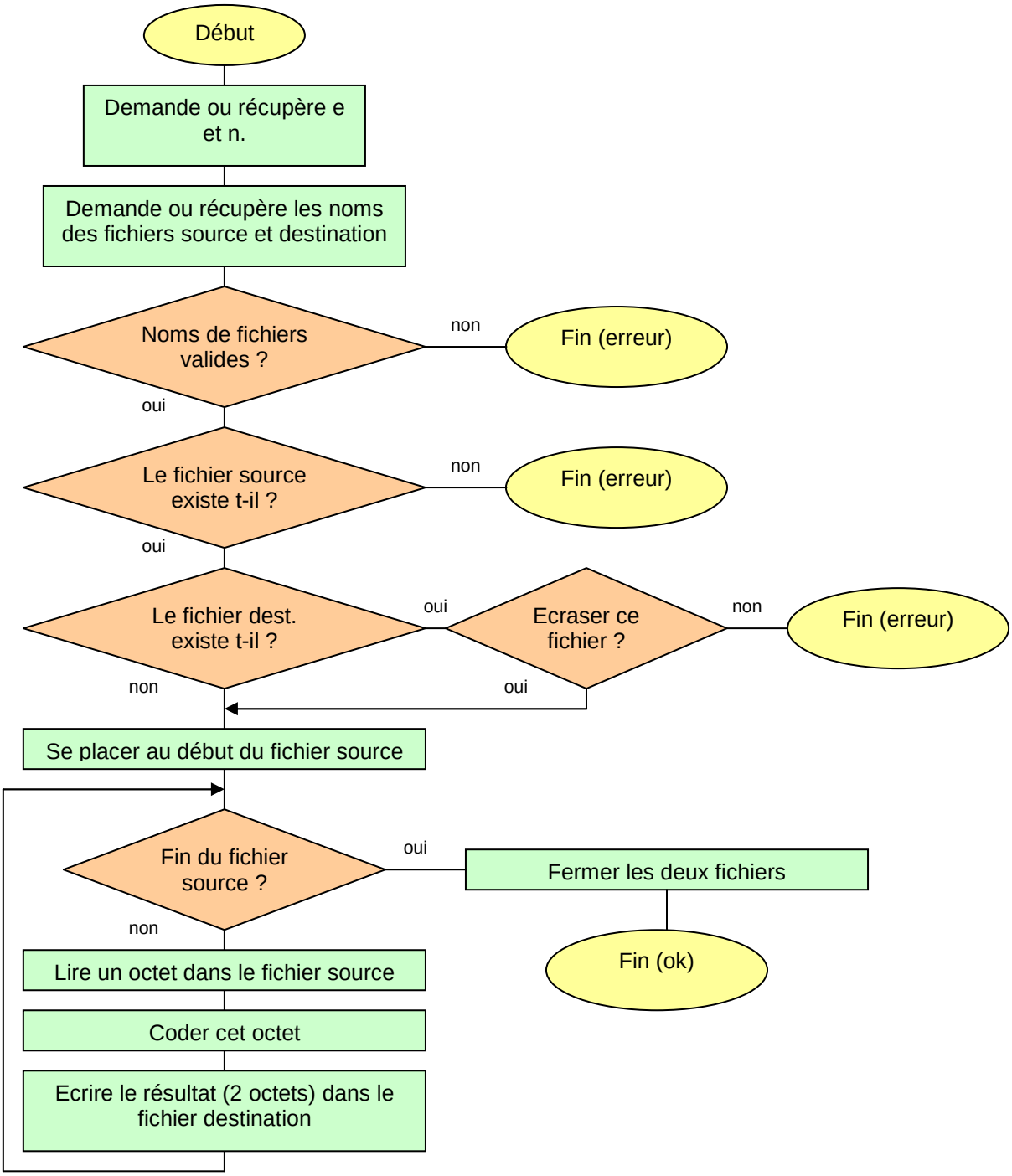
```
codage(lettre : caractère, n : entier long , e : entier long ; c : entier
court)
$1
      entiers courts : i, c, t

      t = (entier court)lettre
      c = 1;

      // effectue c = t^e mod n
      POUR i <- 0 JUSQU'A i < e FAIRE
      $11
            c <- c * t
            c <- reste de c modulo n
            i <- i+1
      11$

      RENVOYER c
1$
```

On peut remarquer deux choses :

- Le caractère `lettre` (8 bits) est converti au format « entier court » 16 bits au moyen d'un transtypage.

- La fonction puissance est réalisé au moyen d'une boucle de multiplication. Toutefois, à chaque cycle, on ne multiple que le reste modulo n de la multiplication précédente. Cela est possible car $\mathbf{a}^{\,p}\ [n] = \mathbf{a}^{\,p-1}\ [n] \times \mathbf{a}$. Cela permet d'économiser un temps de calcul considérable et d'empêcher des débordements de mémoire.

A cet algorithme de codage, nous devons ajouter un certain nombre de protections et de vérifications concernant l'ouverture des fichiers. Appelons « fichier source » le fichier à coder, et « fichier destination » le fichier une fois crypté.



3) Décryptage

Le décryptage fonctionne comme le cryptage, à trois détails près :

- L'algorithme de décodage devient $\mathbf{t_i = c_i^{\,d}\ [n],}$
- Le fichier source devient le fichier crypté, et le fichier destination le fichier une fois décrypté,
- Lors du décodage, on lit 2 octets à la fois dans le fichier source, et en décryptant, on obtient un caractère codé sur 1 octet.

Voici donc notre algorithme de déchiffrement :

```
decodage(lettre : entier court, n : entier long , d : entier long ; c :
caractère)
$1
      entiers courts : i, c
      caractère : t

      // effectue c = t^e mod n
      POUR i <- 0 JUSQU'A i < d FAIRE
      $11
            c <- c * lettre
            c <- reste de c modulo n
            i <- i+1
      11$

      t = (caractère) c

      RENVOYER t

1$
```

4) Forçage

L'algorithme de forage est extrêmement simple : il s'agit principalement de deux boucles imbriquées, qui vont tester toutes les combinaisons de **p** et **q** inférieurs à 255 afin de déterminer le couple dont le produit donne **n**. La suite nous ramène en terrain connu : on recalcule **z**, puis enfin **d**. 0 est renvoyé si **d** n'a pas pu être trouvé.

```
forcage(n : entier long, e : entier long ; d : entier long)
$1
      entiers longs : p, q, d, z

      //boucle de recherche systématique de p et q
      POUR p <- 3 JUSQU'A p < PQ_MAX FAIRE
      $11
            POUR q <- 3 JUSQU'A q < PQ_MAX FAIRE
            $111
                  //si p et q correspondent, c'est gagné !
                  SI n = p*q ALORS
                  $1111
                        INTERROMPRE
                  1111$
                  q <- q + 2
            111$
            SI n = p*q ALORS
            $112
                  INTERROMPRE
            112$
            p <- p + 2
      11$

      //si on a pas trouvé p et q, quitte
      SI n ≠ p*q ALORS
      $12
            RENVOYER 0
      12$
      z <-(p-1)*(q-1)

      d <- 0
      //calcule d : trouve l'inverse de e modulo z

      TANT QUE reste de (e*d) modulo z ≠ 1 FAIRE
      $13
            d <- d + 1
            //anti boucle infinie, s'il n'y a pas d'inverse
            SI d > PQ_MAX * PQ_MAX ALORS
                  RENVOYER 0
      13$
      RENVOYER d
1$
```

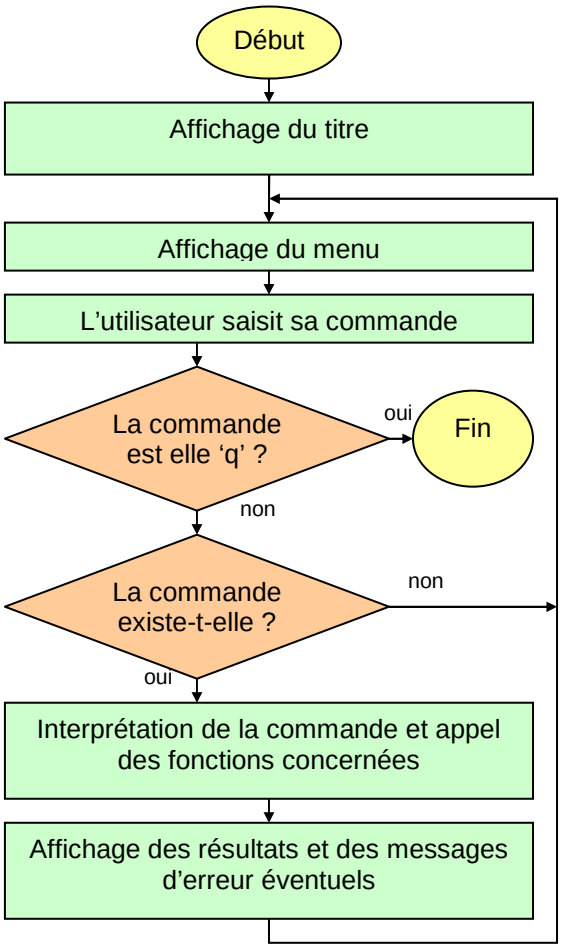
Remarques :

- Lors du calcul de **p** et **q**, on incrémente **p** et **q** de 2 à chaque cycle, car ils sont premiers, et ne peuvent donc pas être pairs. Cela divise par 4 le temps nécessaire pour déterminer **p** et **q**.
- Une protection a été ajoutée lors du calcul de **d**. En effet, dans le cas où le **e** entré serait incorrect, il pourrait n'y avoir aucun inverse de e modulo n.

5) Interface Homme Machine, gestion du temps et des erreurs

a) IHM

Le dialogue entre l'utilisateur et la machine se fait au moyen d'une boucle classique :



Le menu contient les éléments suivants :

```
[k] générer les clés publiques et privées
[c] crypter un fichier
[d] décrypter un fichier
[f] pour lancer le module de forçage
[q] quitter
```

Nous ne rentrerons pas dans le détail pour le code de ce menu, car ce n'est pas l'objet du projet, et car il ne comporte pas de difficulté particulière.

b) Gestion du temps

Les principales fonctions (génération des clés, cryptage, décryptage et forçage) on été pourvues d'un système de chronométrage, afin de pouvoir apprécier le temps d'exécution des différents calculs. Son fonctionnement est extrêmement simple : on mémorise l'heure système juste avant d'exécuter l'opération, puis on soustrait cette valeur à celle de l'heure à laquelle se termine l'opération. Nous obtenons une différence qui représente la durée du calcul, à la seconde près.

c) Gestion des erreurs

Chaque fonction renvoie un code d'exécution, indiquant si celle-ci s'est déroulée correctement. Cela permet à notre programme d'informer l'utilisateur si une opération s'est déroulée sans souci, ou non.

Tests et performances

Dans cette partie, nous allons détailler le comportement de notre programme pour des clés de différentes tailles, et nous tenterons de dresser un bilan de la complexité et de la sécurité des algorithmes composant le RSA.

Tout d'abord, il est très difficile d'estimer la durée de calcul des clés, ainsi que la durée de forçage pour des valeurs de p et q < 255. En effet, ces calculs se font quasiment instantanément. Afin de pouvoir avoir des résultats significatifs, nous avons porté la constante **PQ_MAX** à 65535, ce qui est le maximum physiquement acceptable par le programme. Nous obtenons alors des clés allant jusqu'à 4 milliards, beaucoup plus longues à calculer. Il faut par contre signaler que, bien que les clés soient correctes, le cryptage/décryptage ne fonctionne plus correctement, du fait d'un débordement de capacité.

1) Génération des clés

Celle-ci se fait en trois étapes. La génération de **p** et **q** est quasiment instantanée, tout comme celle de **e**. L'opération prenant le plus de temps est le calcul de **d**. Ce calcul peut s'effectuer en des temps très variables, ne dépendant absolument pas de **n**. Prenons l'exemple suivant :

n	465 236 741	155 744 969
e	25 261	13 313
d	62 738 725	179 086 157
Temps de calcul	4 secondes	11 secondes

Nous remarquons que le deuxième calcul, bien qu'utilisant des nombres **n** et **e** plus petits, a duré presque trois fois plus longtemps que le premier.

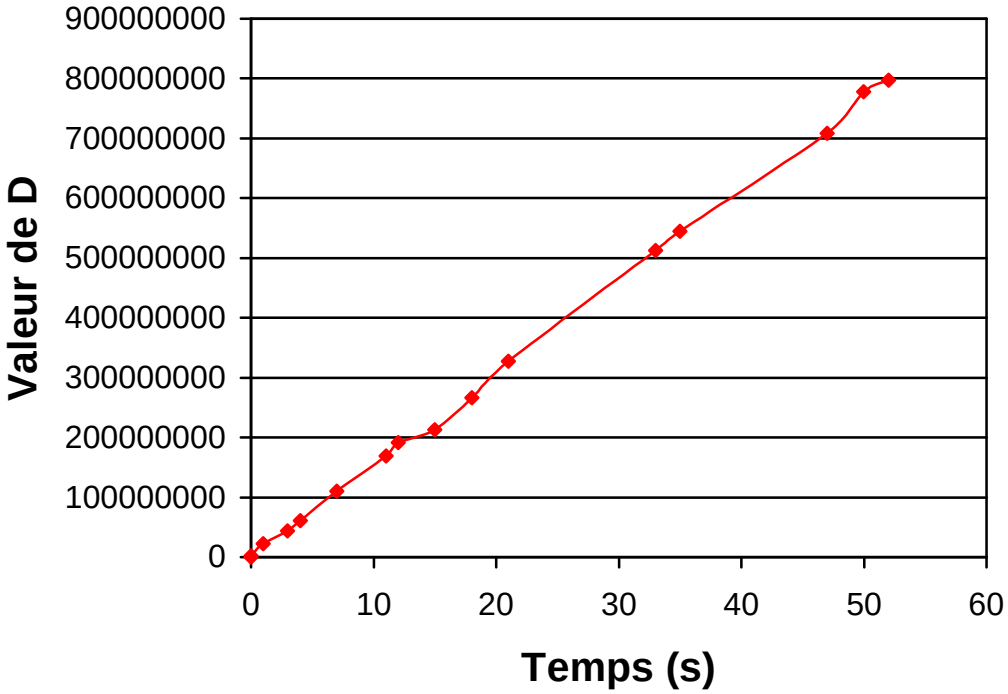
La vitesse de génération des clés ne dépend donc pas de **n**, mais plutôt de **d**. En effet, plus celui-ci est élevé, plus le calcul sera long. Cela s'explique par l'algorithme utilisé, qui sera d'autant plus long que **d** est grand.

Nous pouvons donc dresser des statistiques du temps de calcul en fonction de d :

D=	323603	1960111	22607593	43389697	60954633	110268537	169580963	191413867
T(s)	0	0	1	3	4	7	11	12

D=	212757921	266988069	327264819	51214311	544384595	708356463	778516771	1870274413
T(s)	15	18	21	33	35	47	50	132

A titre indicatif, ces valeurs on été mesurées sur un PC Intel Pentium III 650 MHz



Nous pouvons facilement remarquer que cette courbe est à peu près proportionnelle. Notre algorithme de calcul des clés est donc à **complexité linéaire** (cela se vérifie en analysant le code source).

2) Forçage des clés

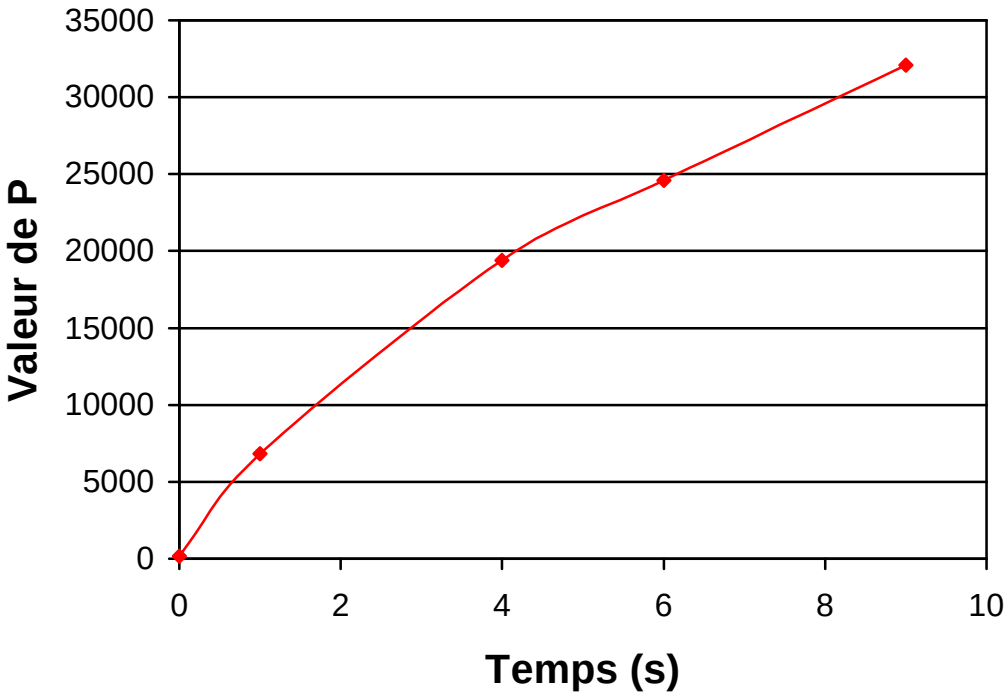
L'opération de forçage des clés se déroule elle aussi en trois étapes :

- Tout d'abord, déterminer **p** et **q**
- Ensuite, déterminer **z** (opération instantanée)
- Enfin, calculer **d** (de la même façon que pour la génération des clés)

Pour déterminer **p** et **q** à partir de **n**, il faut tester systématiquement les combinaisons de couples (p,q), pour trouver celle dont le produit fait **n**. Nous limitons p et q à une certaine valeur maximum, de la même façon que lors du cryptage. Appelons cette valeur **PQ_MAX**.
Pour p et q inférieurs à PQ_MAX, il existe donc $(PQ\_MAX)^2$ couples (p , q) possibles. Lorsqu'on double PQ_MAX, on quadruple de nombre de couples possibles, et donc le temps potentiel de calcul de p et q. L'algorithme de forçage est donc à **complexité polynomiale**. Il faut en plus lui ajouter le calcul de d, qui est de complexité linéaire.

Toutefois, la structure de ce dernier fait qu'il est très rapide de casser une clé dont la composante p serait petite (et cela même si q est grand). Nous allons donc donner quelques exemples de forçage de clés, pour lesquels p et q sont de même ordre de grandeur.

N	36051	37132441	419579269	649705789	897386827
P	183	6827	19767	24611	32069
Q	197	5483	19381	26399	27983
T(s)	0	1	4	6	9



Nous obtenons ici une relation quasiment linéaire, ce qui est tout a fait normal, car nous n'avons pas fait varier la valeur de PQ_MAX (fixée à 65535 pour tous ces tests).

Pour optimiser le forçage, il faut donc choisir PQ_MAX le plus faible possible, mais bien évidemment supérieur à max(p,q).

3) Cryptage / décryptage

Ces opérations dépendent de deux paramètres :

- La taille du fichier à traiter
- La taille de **e** pour le codage, et **d** pour le décodage.

En observant ces algorithmes, nous pouvons voir qu'ils ont une complexité linéaire, selon ces deux paramètres. Nous ne pourrons toutefois pas le prouver au moyen de valeurs relevées, car le cryptage et décryptage ne fonctionnent correctement que pour des valeurs très faibles de p et q. Dans ce cas, le temps d'exécution dépasse rarement la seconde, et n'est donc pas significatif.

4) Conclusion

Soient :

- $\mathbf{T_D}$ le temps de calcul de **d** (qui est quasiment égal au temps total de calcul des clés)
- $\mathbf{T_{Cr}}$ le temps nécessaire pour crypter un fichier
- $\mathbf{T_{Décr}}$ le temps nécessaire pour le décrypter
- $\mathbf{T_{PQ}}$ le temps nécessaire pour trouver **p** et **q** à partir de **n**.

Reprenons notre exemple ci-dessus :

- Pour générer les clés, puis à la fin décrypter le message, George va utiliser le temps $\mathbf{T_D + T_{Décr}}$

- Pour crypter le fichier, Saddam va utiliser le temps $\mathbf{T_{Cr}}$

- Pour forcer les clés et décrypter le message, Vladimir va utiliser le temps $\mathbf{T_{PQ} + T_D + T_{Décr}}$

Vladimir va donc passer $\mathbf{T_{PQ}}$ de temps en plus que George pour décrypter le message de Saddam. C'est pour cela que toute la sécurité du RSA repose sur le temps nécessaire pour calculer p et q à partir de n.

Pour conclure, on peut constater que le codage RSA est viable, car le temps de génération des clés augmente linéairement, alors que le temps de forçage de ces clés est suivant une parabole. Cela veut dire que plus on utilise des nombres grands, et plus la différence de temps sera significative entre le calcul des clés et le forçage du code.

C'est ainsi que sur des clés de 1024 bits, on obtient souvent une différence de l'ordre de plusieurs semaines, ou plusieurs mois ! Ainsi George aura le temps de prendre toutes les dispositions nécessaires avant que Vladimir n'ai pu avoir connaissance du fameux message.

Conclusion

Ce programme de cryptage nous a permis d'avoir une nouvelle approche de la conception, où le temps la réflexion et les études préalables sont bien plus importantes et difficiles que la programmation en elle-même.

Nous avons également pu combiner les mathématiques et l'informatique dans un domaine concret et particulièrement d'actualité : la cryptographie.

Enfin, ce projet nous a montré la nécessité d'optimiser nos algorithmes afin d'avoir des programmes plus performants.

Ce projet est donc un petit pas pour la cryptographie, mais un grand pas dans notre formation !

Les auteurs tiennent à remercier George, Saddam et Vladimir, sans qui ce rapport n'aurait pas été ce qu'il est actuellement !