# Implémentation du cryptosystème RSA

Introduction à la cryptographie, TP du 12 décembre 2008
`www-fourier.ujf-grenoble.fr/~eiserm/cours#crypto`

## OBJECTIFS

Dans ce projet on se propose d'implémenter le cryptosystème RSA.

Les programmes sont à rédiger en C/C++, ou Java, ou tout autre langage avec accord préalable. Les exemples explicités ci-dessus sont écrits en C++. Leur traduction en d'autres langages devrait être facile mais prend un peu de temps.

On accordera un soin particulier à la rédaction du code source :
– Idéalement l'implémentation se fait dans un fichier source unique.
– Commencer par les commentaires usuels : auteur(s), date, objectifs.
– Commenter chaque fonction par sa spécification (pré- et postconditions).
– Veiller à la lisibilité : noms parlants, style cohérent, bonne indentation, ...
– Vos réponses aux questions annexes peuvent être jointes en commentaire.

Le fichier final — abondamment testé, vérifié, relu — est à envoyer jusqu'au vendredi 19 décembre 2008 à `Michael.Eisermann@ujf-grenoble.fr`

## 1. CONVENTIONS ET PRÉPARATIONS

1.1. **Fichiers.** Pour commencer, vous pouvez créer un sous-répertoire `rsa` et copier les fichiers modèles et exemples dans votre répertoire :

```
mkdir rsa;  cd rsa;  cp -v /home/P/eiserm/rsa/* .
```

Vous y trouverez les exemples `test*.rsa`. Le fichier source `rsa.cc` fournit des modèles à compléter dans la suite. Tous les fichiers sont disponibles en ligne sous `~eiserm/Enseignement/crypto/tp-rsa/`.

1.2. **Grands entiers.** On supposera dans la suite que la classe `Integer` implémente les nombres entiers, sans restriction de taille outre la mémoire disponible. En Java on pourra prendre la classe `BigInteger`, en C++ on utilisera la classe `mpz_class` de la bibliothèque GMP (GNU Multiple Precision Library).

1.3. **Division euclidienne.** On rappelle que pour tout $a, b \in \mathbb{Z}$ où $b \neq 0$ il existe une unique paire $(q, r) \in \mathbb{Z} \times \mathbb{Z}$ telle que $a = bq + r$ et $0 \leq r < |b|$. Dans ce cas on écrit $q = a \operatorname{quo} b$ et $r = a \operatorname{rem} b$. Pour $a \geq 0$ ceci est implémenté en C/C++ et en Java par les opérateurs `a/b` et `a%b` ; pour $a < 0$, par contre, le résultat est l'unique paire $(q', r')$ tel que $a = bq' + r'$ où $-|b| < r' \leq 0$. La fonction

```
Integer pmod( const Integer& a, const Integer& m )
```

remédie à cet inconvénient et renvoie correctement $a \operatorname{rem} m$.

1.4. **Nombres aléatoires.** Nous aurons besoin d'un générateur de nombres aléatoires `random(min,max)` afin de tirer un entier dans l'intervalle [`min`, `max`].

En C++ on pourra faire appel à la bibliothèque GMP :

```
Integer random( const Integer& min, const Integer& max )
{
  static gmp_randclass generateur(gmp_randinit_default);
  return generateur.get_z_range(max-min+1) + min;
}
```

D'autres langages fournissent des fonctions `random(min,max)` similaires.

1.5. **Chronométrage.** Le fichier `timer.hh` implémente un chronométrage suffisamment fin pour nos besoins. La classe `Timer` s'utilise simplement par

```
Timer timer;
timer.start();
// insérer le calcul ici
timer.stop()
cout << "temps écoulé " << timer << endl;
```

D'autres langages fournissent des fonctionnalités similaires.

## 2. Puissance modulaire

2.1. **Implémentation.** Implémenter deux fonctions efficaces

```
Integer puissance( Integer base, Integer exp );
Integer puissance( Integer base, Integer exp, const Integer& mod );
```

La première calcule $b^e$ pour $b, e \in \mathbb{Z}$, $e \geq 0$. La deuxième calcule $b^e \operatorname{rem} m$ pour $b, e, m \in \mathbb{Z}$, $e \geq 0$. (Comment traiter le cas $m = 0$ ? puis le cas $b < 0$ ?)

2.2. **Performance.** La fonction `test_puissance` calcule $3^m \operatorname{rem} m$ à l'aide de votre fonction `puissance` ci-dessus pour $m = 10^{2^k}$ où $k = 0, 1, 2, \ldots$. Jusqu'où pouvez-vous aller, environ ? Discutez puis expliquez le temps d'exécution observé.

2.3. **Plausibilité.** Quels sont les résultats des calculs précédents ? Vérifier par un raisonnement mathématique que les résultats trouvés sont corrects. Pour cela déterminer la structure du groupe $\mathbb{Z}/m^\times$ des éléments inversibles modulo $m = 10^j$.

## 3. Inverse modulaire

3.1. **Implémentation.** Implémenter une fonction

```
Integer inverse( Integer a, const Integer& mod );
```

qui calcule efficacement l'inverse de $a$ modulo $m$, si $a$ est inversible, et qui renvoie 0 sinon. Ici on peut supposer que $m \geq 1$. S'il existe, l'inverse de $a$ modulo $m$ sera représenté par l'unique entier $u \in \mathbb{Z}$, $0 < u < m$, vérifiant $au \equiv 1 \mod m$.

3.2. **Plausibilité.** Testez votre implémentation avec la fonction `test_inverse`. Elle choisit deux entiers aléatoires à $2^k$ bits, c'est-à-dire $a, m \in \{n, \ldots, 2n\}$ pour $n = 2^{2^k}$ où $k = 3, 4, 5, \ldots$, puis elle calcule l'inverse de $a$ modulo $m$ à l'aide de votre fonction `inverse` ci-dessus ; si l'inverse $u$ existe, elle vérifie si $au \equiv 1 \mod m$. Pourquoi $a$ est-il assez souvent inversible modulo $m$ ?

3.3. **Performance.** Dans les calculs précédents, jusqu'à quelle valeur de $k$ pouvez-vous aller, environ ? Discutez puis expliquez le temps d'exécution observé.

## 4. Le théorème chinois

Bien que ce ne soit pas nécessaire pour implémenter RSA, on fait une petite excursion pour implémenter la partie intéressante du théorème des restes chinois.

4.1. **Implémentation.** Implémenter efficacement le théorème des restes chinois :

```
Bienvenue au théorème des restes chinois !  SVP, entrez des modules
m_1, m_2, ..., premiers entre eux, ainsi que des valeurs y_1, y_2, ...
Je calculerai le plus petit entier x >= 0 qui soit congru à y_i mod m_i.
Jusqu'ici la valeur cherchée vaut 0 modulo 1.
Entrez le module m_1 svp : 5
Entrez la valeur y_1 svp : 2
Jusqu'ici la valeur cherchée vaut 2 modulo 5.
Entrez le module m_2 svp : 6
Entrez la valeur y_2 svp : 4
Jusqu'ici la valeur cherchée vaut 22 modulo 30.
Entrez le module m_3 svp : 7
Entrez la valeur y_3 svp : 4
Jusqu'ici la valeur cherchée vaut 172 modulo 210.
Entrez le module m_4 svp : 0
Les modules ne sont plus premiers entre eux. Au revoir.
```

Cette fonction demande à l'utilisateur des paires d'entiers $(m_1, y_1), \ldots, (m_k, y_k)$ et calcule le plus petit $x \geq 0$ vérifiant $x \equiv y_i \mod m_i$ pour tout $i = 1, \ldots, k$. La fonction s'arrête si $m_1, m_2, \ldots, m_k$ ne sont plus premiers entre eux. Sinon elle continue à demander $(m_{k+1}, y_{k+1})$. Pour ce calcul on utilisera la fonction `inverse`. On pourra utiliser une boucle et éviter les tableaux/vecteurs.

4.2. **Illustration.** Une bande de 17 pirates possède un trésor constitué de pièces d'or d'égale valeur. Ils projettent de les partager également, et de donner le reste au cuisinier chinois. Celui-ci recevrait alors 3 pièces. Mais les pirates se querellent, et six d'entre eux sont tués. Un nouveau partage donnerait au cuisinier 4 pièces. Dans un naufrage ultérieur, seuls le trésor, six pirates et le cuisinier sont sauvés, et le partage donnerait alors 5 pièces d'or à ce dernier. Quelle est la fortune minimale que peut espérer le cuisinier s'il décide d'empoisonner le reste des pirates ?

4.3. **Performance.** Trouver le plus petit entier $x \geq 0$ tel que $x \equiv 2 \mod 10^{18}$ et $x \equiv 3 \mod 1234567890123456789$. Le calcul est-il assez rapide ?

## 5. CRYPTAGE RSA

On se propose de coder un *texte* par la méthode RSA. Pour cela il nous faut d'abord transformer un texte en entier, puis retransformer un entier en texte.

On interprète les lettres `A`, ..., `Z` comme nombres $1,\dots,26$ et ' _ ' comme 0. Ces transformations sont effectuées par les fonctions

```
Integer char_to_integer( char c );
char integer_to_char( Integer n );
```

On peut ainsi interpréter tout texte $t = (c_k, c_{k-1}, \dots, c_1, c_0)$ comme l'entier $n = c_k b^k + c_{k-1} b^{k-1} + \cdots + c_1 b^1 + c_0$, et réciproquement tout entier $n$ comme un texte $t$. Ayant fixé ces conventions, le texte $t =$ `ABC` correspond au nombre $n = 1 \cdot 27^2 + 2 \cdot 27 + 3 = 786$.

5.1. **Codage.** Écrire deux fonctions efficaces

```
Integer convert( const string& texte );
string convert( Integer nombre );
```

qui effectuent la transformation entre texte et entier. On utilisera la méthode de Horner pour la première, pour la deuxième on itère division euclidienne par 27. (La fonction `renverser(vec)` pourra vous être utile.)

Dans le même style, écrire deux fonctions efficaces

```
Integer convert( const vector<Integer>& chiffres, const Integer& base );
vector<Integer> convert( Integer nombre, const Integer& base );
```

qui effectuent ces transformations dans n'importe quelle base $b \geq 2$.

5.2. **Cryptage.** Écrire une fonction efficace

```
cryptageRSA( const Integer& mod, const Integer& exp, string& texte );
```

qui crypte un texte selon la méthode RSA : le texte $t$ en base 27 est transformé en un entier $n$, puis $n$ est développé en $n_k, \dots, n_0$ en base $m$. Ainsi on peut appliquer RSA pour calculer $n'_i \leftarrow n_i^e \operatorname{rem} m$. On compose $n'_k, \dots, n'_0$ en base $m$ pour obtenir l'entier crypté $n'$, puis le texte crypté $t'$.

5.3. **Test.** La fonction `cryptageRSA( istream& in, ostream& out )` lit les données $m$, $e$, $t$ du flot d'entrée puis écrit $m$, $e$, $t'$ dans le flot de sortie. Si votre programme compilé s'appelle `rsa`, alors la commande `./rsa < test1.rsa` affichera `15 3 FELICITATIONS`. Décryptez ensuite `test2.rsa` et `test3.rsa`.

5.4. **Casser un cryptage trop faible.** Essayez de décrypter le message dans le fichier `test4.rsa`, qui contient la clef publique $(m, e)$ et le message crypté $t'$. Pour la décomposition en facteurs premiers utilisez un logiciel de votre choix.

**Avertissement.** *Le cryptage des textes n'est choisi ici qu'à titre d'illustration. Dans une application réaliste on ne crypte pas des messages tels quels avec la méthode RSA. On échange plutôt un secret auxiliaire de taille relativement petite via RSA. Ce secret sert ensuite de clef pour une méthode de cryptage à clef secrète, souvent moins coûteuse que RSA pour des grandes quantités de données.*

## 6. Test de primalité

6.1. **Implémentation.** Écrire une fonction efficace

```
bool estPseudoPremier( const Integer& n, const Integer& x,
                       const Integer& q, const Integer& e );
```

qui teste si $n$ est pseudo-premier à base $x$, dans le sens de Miller-Rabin. Ici $n-1=2^e q$, $e \geq 1$, $q \geq 1$, et $q$ est impair. En déduire une fonction efficace

```
bool estProbablementPremier( Integer n, int essais= 50 );
```

qui effectue le test de Miller-Rabin jusqu'à 50 fois, disons. Il sera utile d'attraper quelques cas exceptionnels (entiers négatifs, puis $0,1,2,3$, puis les nombres pairs).

6.2. **Plausibilité.** La fonction `test_MillerRabin` trouve le plus petit $a \in \mathbb{N}$ tel que $10^k + a$ soit premier, où $k = 10, 20, 30, \ldots$. Peut-on faire confiance à ces résultats probabilistes ? Quelles sont les sources d'erreurs les plus inquiétantes ? Vous pouvez comparer vos résultats entre vous ou avec un logiciel de votre choix.

6.3. **Performance.** Discutez puis expliquez le temps d'exécution observé. Jusqu'à quel valeur de $k$, environ, cette approche vous semble-t-elle assez rapide. Que se passe-t-il quand vous augmenter de 50 à 100 essais ? Testez-le puis expliquez vos observations. Et quand vous diminuez à très peu d'essais ?

## 7. Production de clefs

Jusqu'ici on sait appliquer le cryptage/décryptage à une clef $(m,e)$ et un texte donnés. Ce dernier paragraphe implémente la production de clefs.

7.1. **Nombres premiers aléatoires.** Écrire une fonction efficace

```
Integer premierAleatoire( Integer pmin, Integer pmax );
```

qui produit un nombre premier aléatoire entre $p_{\min}$ et $p_{\max}$.

**Avertissement.** *Nous n'entrons pas dans une discussion détaillée ici et choisissons nos nombres premiers de manière aléatoire. Avec un peu de malchance ceci mènera à un cryptage faible, c'est-à-dire, trop facile à casser. (Pour le moment, fermons les yeux et croisons les doigts…) Pour toute application réaliste la production des nombres premiers « forts » est un sujet important.*

7.2. **Production de clefs.** En déduire une fonction efficace

```
void produireClefRSA( int bits, Integer& n, Integer& c, Integer& d );
```

qui produit une clef aléatoire pour le cryptage RSA. Ici $n$ sera composé de deux facteurs premiers de la taille spécifiée. Avec ces clefs aléatoires, tester le cryptage/décryptage sur quelques exemples.

7.3. **Signature.** Fabriquez ainsi une clef publique $(n,c)$ que vous estimez sûre. Motivez vos choix. Pour signer votre travail, cryptez avec votre clef secrète le message clair $t =$ `NOUS_AVONS_TERMINE_BONNES_VACANCES` (plus vos noms) en un message crypté $t'$. Je vérifierai votre signature à partir des données $n,c,t'$ jointes à votre code source. (Vous pouvez le tester avant l'envoi.)