Université Toulouse III – Paul Sabatier

IUT 'A' Paul Sabatier

Département Informatique

réalisé dans le cadre du DUT Informatique (en Année spéciale)

Mathématiques AS 2018-2019

Programmation des trois algorithmes Rho de Pollard pour la décomposition en produit de deux facteurs premiers spécifiques au cryptosystème RSA

Dossier de modélisation - Juin 2019

Version du 03/06/2019

Auteur
Ludwig Chieng

Tuteur
M. Abdel-Kaddous Taha

Table des matières

Abstract

Ce dossier, dit “de modélisation”, rend compte de l’efficacité des algorithmes Rho de Pollard conçu pour décomposer un produit de deux nombres premiers. Nous exposons tout d’abord un bref point théorique sur les méthodes de Pollard et enfin nous comparons la vitesse d’exécution des trois algorithmes implémentés en Java.

1. Introduction

Les trois algorithmes Rho de Pollard[1] sont trois versions d’une méthode pour factoriser un nombre composé de deux facteurs premiers.

Le cryptosystème RSA[2] repose sur un chiffrement asymétrique où la clef privée peut être calculée en factorisant le module de chiffrement contenu dans la clef publique.

Aujourd’hui, en informatique, de nombreux protocoles implémentent le chiffrement RSA pour sécuriser les communications. Les enjeux soulevés par les algorithmes de Pollard sont donc importants.

2. Rappels théoriques

2.1. La génération des clés privée et publique pour le chiffrement RSA

On pose n produit de p et q deux nombres premiers distincts et m la valeur de la fonction indicatrice d’Euler[3] φ en n.

La clef publique contient le module de chiffrement n et l’exposant de chiffrement e, avec e premier avec $\varphi(n)$.

La clef privée est définie comme le couple (d,n), avec d, l’inverse de e modulo n, avec e relativement premier avec $\varphi(n)$.

La génération des clés privée et publique est réalisée conjointement car c’est en créant la clef privée que l’on engendre la clef publique.

On choisit d’abord p et q différents et premiers. On détermine ensuite n et m de la manière suivante :

$$n = pq \tag{1}$$

$$m = \phi(n) = (p-1)(q-1) \tag{2}$$

On choisit un exposant e tel qu’il soit premier avec $\varphi(n)$ et plus petit que celui-ci.

On calcule d grâce à l’identité de Bézout[4] sachant que :

$$ed \equiv 1 \mod p \tag{3}$$

2.2. Application de la méthode Rho de Pollard pour le RSA

L’algorithme de Pollard a pour ambition de déterminer la clef privée à partir de la clef publique, en trouvant p ou q à partir de n en un temps de calcul raisonnable.

Remarquons que si on trouve l’un, on peut facilement obtenir l’autre grâce à (1), connaissant n.

Malheureusement, passer au crible chacun des entiers compris dans $[2, n/2]$ en espérant trouver un facteur p de n, n’est pas une solution viable pour les très grands n. En effet, le temps d’exécution de ce procédé augmente proportionnellement à n.

Le temps d’exécution moyen des méthodes de Pollard augmente proportionnellement à la racine carré de n. C’est ce que nous allons tenter d’observer.

2.3. Principe des algorithmes de Pollard

Soit l tel que l soit un multiple de $p-1$. On a donc,

$$l = k(p-1),\ k \in \mathbb{N}$$
$$\Leftrightarrow a^l \equiv a^{k(p-1)} \mod p \tag{4}$$

avec a, un entier quelconque. Or, p est premier, donc le petit théorème de Fermat donne,

$$a^{p-1} \equiv 1 \mod p,\ \forall a \in \frac{\mathbb{Z}}{p\mathbb{Z}} \tag{5}$$

Par conséquent,

$$a^l \equiv 1 \mod p \tag{6}$$

D’où, p divise $a^l - 1$.

Or, p divise également n. En supposant que p nous est inconnu, on peut le retrouver en cherchant le PGCD de n et $a^l - 1$.

On se place dans $E = \mathbb{Z}/n\mathbb{Z}$. Soit une fonction $f : E \to E$ donnant des valeurs pseudo-aléatoires, et soit la suite,

$$x_{n+1} = f(x_n) \tag{7}$$

Dans la version la plus simple de l’algorithme de Pollard, à chaque itération, on calcule $pgcd(n, x_n)$. Si on tombe sur un pgcd non trivial, on obtient p.

3. Distinctions entre les trois variantes de la méthode de Pollard

3.1. L’algorithme Pollard 1

Pour optimiser l'exécution, on sera amené à calculer $pgcd(n, x_n - u)$ avec u, un entier quelconque.

On choisit f tel que,

$$x_{n+1} = {x_n}^2 + a \mod n \tag{8}$$

avec a, un entier quelconque.

```
# Pseudo-algorithme de Pollard 1
int n, p, x, x0, a, u;
int d = 1;

input n;
x0 = random(2,n-1);
a = random(1,n-1);
x = x0;
u = x;
forever:
   for(int j=d+1; j < 2*d; j++):
      x = (x^2 + a) % n;
      p = pgcd( n, (x-u)%n );
      if(p > 1 && p != n):
         return p;
   u = x;
   d = d*2;
```

3.2. L'algorithme Pollard 2

Cette version de la méthode de Pollard implémente la même logique que l'algorithme de détection de cycle de Floyd[5]. Cet algorithme dit "du lièvre et de la tortue" consiste à placer deux pointeurs sur une suite (x_n) dont une valeur donnée ne dépend que de la précédente. Les deux pointeurs se déplacent le long de la suite à des vitesses différentes.

Dans notre cas, le premier pointeur est x, le second est y. Pour chaque itération, $x_{n+1} = f(x_n)$ alors que $y_{n+1} = f \circ f(y_n)$. On fait partir x et y du même point, à savoir : x_0.

```
# Pseudo-algorithme de Pollard 2
int n, p, x, x0, a, y;

input n;
x0 = random(2,n-1);
a = random(1,n-1);
x = x0;
y = x;
forever:
   x = (x^2 + a) % n;
   y = (y^2 + a) % n;
   y = (y^2 + a) % n;
   p = pgcd( n, (y-x)%n );
   if(p > 1 && p != n):
      return p;
```

3.3. L'algorithme Pollard 3

Cette variante est dérivée de l'algorithme p–1 de Pollard. A la différence des deux autres, celle-ci ne peut trouver qu'un seul des deux facteurs d'un n donné quelque soit le x_0 de départ. Il est également possible qu'elle échoue, nous y reviendront plus tard.

Tout d'abord, nous devons rappeler quelques notions. Un entier naturel non nul est B-friable si tous ses facteurs premiers sont inférieurs ou égaux à B. Il est B-ultrafriable si tous ses diviseurs primaires sont inférieurs ou égaux à B.

Pour la suite, on appellera B le seuil d'ultrafriabilité minimal pour un nombre donné.

Exemple :

90 est 5-friable car $90 = 2 \times 3^2 \times 5$. Il n'est pas 5-ultrafriable car $3^2 = 9 > 5$. Par contre, il est 9-ultrafriable.

Ayant $n = pq$, avec p et q premiers, on a $p-1$ et $q-1$ possédant un certain seuil de friabilité qu'on notera B_{p-1} et B_{q-1}.

L'algorithme de Pollard 3 est capable de trouver le facteur de n dont le précédent a le seuil de friabilité le plus faible.

```
# Pseudo-algorithme de Pollard 3
int n, p, x, x0;
int i = 2;

input n;
x0 = random(2,n-1);
x = x0;
forever:
   x = x^i % n;
   p = pgcd(n, x-1);
   if(p == n):
      exit(1);
   if(p > 1):
      return p;
   i = i+1;
```

Si $B_{p-1} = B_{q-1}$ alors l'algorithme peut, soit trouver un facteur de n, soit trouver un pgcd égal à n, auquel cas, l'exécution a échoué. Il faut réessayer avec un $x0$ différent.

4. Aperçu de l'application

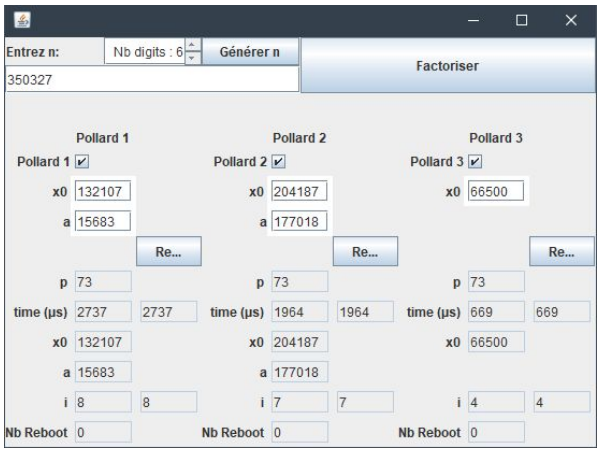
4.1. Environnement de développement et d'exécution

Les trois algorithmes ont été programmé en Java avec l'IDE Eclipse et exécuté avec le JRE 1.8.0_211-b12 sur un processeur à 4 coeurs logiques cadencé à 2,95 GHz en moyenne avec un facteur de charge moyen de 37%.

4.2. Fonctionnalités de l'application

L'interface affiche des zones de saisies pour n et éventuellement, x_0 et a. Le bouton "Factoriser" démarre l'exécution de Pollard 1, puis de Pollard 2 et enfin de Pollard 3. Les résultats sont affichés une fois que les trois algorithmes ont terminé leur exécution.

La zone de saisie de n est doté d'un générateur de n, si besoin. Le nombre de digits de n est paramétrable.



Les résultats d'un algorithme de Pollard contiennent les informations suivantes : le facteur p trouvé, le temps d'exécution en µs, le germe x_0 de la suite, l'éventuel valeur arbitraire a, le nombre d'itérations i du dernier lancement (s'il y en a eu plusieurs), et le nombre de relancement $nb\ Reboot$ de l'algorithme si il y a erreur durant son exécution (ex: pgcd = n).

Un relancement consiste à exécuter l'algorithme de nouveau mais avec un x_0 et/ou un a différent.

5. Mise à l'épreuve des trois algorithmes

5.1. Premières traces d'exécution

Pour cet exemple, on prend $n = 350327$, avec $p = 73$ et $q = 4799$.

```
# Trace d'exécution de Pollard 1
## POLLARD 1   n=350327    x0=132107    a=15683
# Update u=132107 ---
# Update d=2 --------

j=3    2d=4
132107^2+15683=34973
pgcd(350327,253193)=1

# Update u=34973 ----
# Update d=4 --------

j=5    2d=8
34973^2+15683=134855
pgcd(350327,99882)=1
j=6    2d=8
134855^2+15683=61811
pgcd(350327,26838)=1
j=7    2d=8
61811^2+15683=299469
pgcd(350327,264496)=1

# Update u=299469 ---
# Update d=8 --------

j=9    2d=16
299469^2+15683=87606
pgcd(350327,138464)=1
j=10    2d=16
87606^2+15683=213330
pgcd(350327,264188)=1
j=11    2d=16
213330^2+15683=125321
pgcd(350327,176179)=1
j=12    2d=16
125321^2+15683=209314
pgcd(350327,260172)=73
---------------------
p=73
```

```
# Trace d'exécution de Pollard 2
## POLLARD 2   n=350327    x0=204187    a=177018
204187^2+a=91717
(204187^2+a)^2+a=133183
pgcd(350327,41466)=1

91717^2+a=133183
(133183^2+a)^2+a=228581
pgcd(350327,95398)=1

133183^2+a=131843
(228581^2+a)^2+a=341747
pgcd(350327,209904)=1

131843^2+a=228581
(341747^2+a)^2+a=292354
pgcd(350327,63773)=1

228581^2+a=280491
(292354^2+a)^2+a=62410
pgcd(350327,132246)=1

280491^2+a=341747
(62410^2+a)^2+a=326043
pgcd(350327,334623)=1

341747^2+a=224748
(326043^2+a)^2+a=322641
pgcd(350327,97893)=73
---------------------
p=73
```

```
# Trace d'exécution de Pollard 3
## POLLARD 3   n=350327    x0=66500
66500^2=72279
pgcd(350327,72278)=1

72279^3=246228
pgcd(350327,246227)=1

246228^4=335509
pgcd(350327,335508)=73
---------------------
p=73
```

Tableau comparatif des performances des trois algorithmes de Pollard pour une factorisation donnée

$n = 350327$	Pollard 1	Pollard 2	Pollard 3
Temps* (µs)	2737*	1964*	669*
p	73	73	73
x_0	132107	204187	66500
a	15683	177018	-
Nombre d'itérations	8	7	4**
Nombre d'échec	0	0	0

**Pour ce tableau, le temps d'exécution comprend également le temps requis pour réaliser les écritures sur la sortie standard, temps qui n'est pas négligeable. Les prochains temps d'exécution ne les prennent pas en compte.*

***Pour Pollard 3, on initialise i à 2 donc le nombre réel d'itérations correspond en fait à la valeur affichée - 1.*

5.2. Estimation de la fluctuation de la puissance de calcul

Étant donné que nous allons étudier le temps d'exécution en fonction de n, nous allons aussi estimer la fluctuation de la puissance de calcul en fonction de n, par soucis de rigueur.

On évalue la fluctuation en étudiant l'écart-type σ du temps d'exécution de 1000 factorisations pour un n donné et un algorithme donné.

Les 1000 factorisations s'effectuent avec le même x_0 et éventuellement le même a.

Pollard 1		Pollard 2		Pollard 3	
n	σ (en µs)	n	σ (en µs)	n	σ (en µs)
93	47,67	77	10,29	33	20,45
254	68,08	451	2,92	259	2,15
1903	77,72	1507	143,04	1981	5,57
21041	32,78	25351	18,61	51943	9,74
598061	103,58	278723	304,08	197459	28,24
4344731	192,39	9613469	96,74	5974601	137,70

La dispersion du temps d'exécution ne semble pas être lié à la taille de n.

La moyenne des fluctuations de la puissance de calcul est de 72,32 ± 78,42 µs. Par sécurité, on prendra la borne supérieure, donc la fluctuation moyenne de la puissance de calcul est de 151 µs.

5.3. Étude du temps d'exécution en fonction de la taille de n et de l'algorithme choisi

5.3.1. Hypothèse

Le temps d'exécution des algorithmes de Pollard augmente proportionnellement à la racine carrée de n.

C'est-à-dire que si y est le temps d'exécution, et x la taille de n, on a $y = kx^{0.5}$, avec k un coefficient qui ne dépend que de la puissance de calcul du processeur.

5.3.2. Protocole

Échantillon

On dispose d'une liste de 2227 valeurs n qui vont de 111 à 1 604 099 367 841. On a environ 200 valeurs pour chaque puissance de 10.

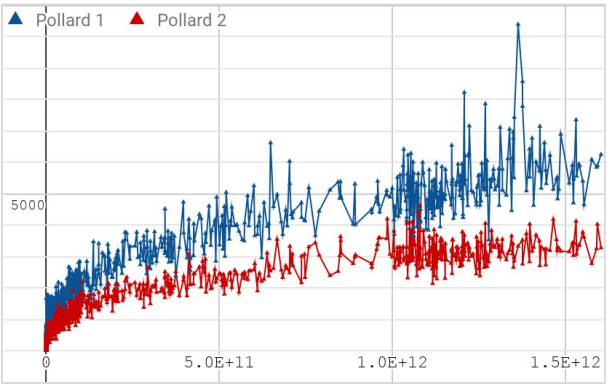
Cela signifie qu'il y a 200 valeurs qui sont de l'ordre de 10^3, de même, il y a 200 valeurs de l'ordre de 10^4, etc...

Procédé

On réalise la factorisation des 2227 valeurs de n pour les trois méthodes. On répète cela 20 fois, en vue de faire la moyenne du temps d'exécution pour minimiser les fluctuations.

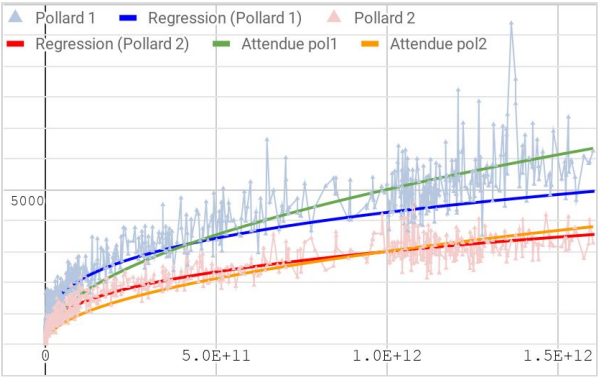
5.3.3. Résultats et interprétations

Temps (en µs) d'exécution (en ordonnées) de Pollard 1 et Pollard 2 en fonction de n (en abscisse)



Les deux courbes pour Pollard 1 et 2 semblent dessiner une croissance, malgré les fluctuations encore importantes même après avoir tenté de lisser la valeur par une moyenne de 20 valeurs.

Régressions de puissance et courbes attendues (théoriques) pour Pollard 1 et 2



— *Régression (Pollard 1) :* $y = 0.7239x^{0.3141}$ $R^2 = 0.927$
— *Régression (Pollard 2) :* $y = 0.1524x^{0.3578}$ $R^2 = 0.963$
— *Attendue (Pollard 1) :* $y = 0.005x^{0.5}$ *k fixé empiriquement*
— *Attendue (Pollard 2) :* $y = 0.003x^{0.5}$ *k fixé empiriquement*

Notons que la fluctuation de puissance de calcul de 151 µs (estimée en 5.2) est petite par rapport au temps d'exécution, pour la plage de n choisie pour l'étude.

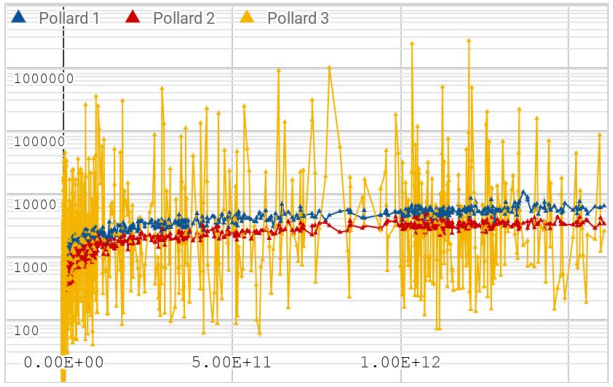
Pour Pollard 2, la courbe attendue (théorique) correspond aux valeurs pratiques. En revanche, pour Pollard 1, il y a un écart plus important entre la théorie et les valeurs mesurées.

Néanmoins, les deux courbes mesurées semblent évoluer de manière proportionnel à la racine carrée de n mais de manière moins prononcée pour Pollard 1 au regard de nos mesures.

Source de l'écart avec la valeur théorique

Les anomalies de mesure sont dues aux fluctuations de la puissance de calcul mais surtout à la variation pseudo-aléatoire lié au choix de x_0 et éventuellement de a.

Temps (en µs) d'exécution (en ordonnées) de Pollard 3 en fonction de n (en abscisse)



L'axe des ordonnées est en échelle logarithmique par soucis de lisibilité du graphique.

Pour Pollard 3, il ne semble pas y avoir de corrélation claire entre temps d'exécution et taille de n.

En tout cas, s'il y a une corrélation, on ne peut pas la détecter car il y avoir un autre facteur qui rentre en jeu et qui rend l'influence de n négligeable devant celle de ce facteur de confusion.

5.4. Conjecture à propos du temps d'exécution de Pollard 3

Comme expliqué précédemment (en 3.3), L'algorithme de Pollard 3 est capable de trouver le facteur de n dont le précédent a le seuil de friabilité le plus faible.

Si $B_{p-1} < B_{q-1}$, alors Pollard 3 trouvera p et non q. La grande majorité de nos mesures ont vérifiées l'observation suivante, cependant, nous ne savons pas comment démontrer, réfuter ou même expliquer entièrement cette assertion.

Observation

Supposons que $B_{p-1} < B_{q-1}$, alors le nombre d'itérations maximal pour trouver p est très souvent égal à B_{p-1}.

Quelques exemples

n	p	q	B_{p-1}	B_{q-1}	Itérations	Temps
1101	367	3	**61**	2	61	919
385789	1217	317	**19**	79	19	85
524851	157	3343	13	557	4	63
2500567	1949	1283	**487**	641	487	1027
2512793	1663	1511	277	**151**	151	344
3476461	1823	1907	**911**	953	911	4378
3513751	1951	1801	13	**5**	5	33

On note des cas particuliers pour certaines combinaisons de n et x_0, le nombre d'itérations est plus faible.

Étant donné que le nombre d'itérations est le principal élément qui semble influer sur le temps d'exécution, et qu'il est indépendant de la taille de n, on comprend les fluctuations aléatoires du dernier graphique.

Conclusion

L'application programmée en Java permet de rendre compte du fonctionnement des trois algorithmes de Pollard pour la factorisation d'un nombre composé de deux facteurs premiers. Avec celle-ci, nous avons pu étudier le temps d'exécution des trois méthodes.

L'hypothèse posée indiquait que le temps d'exécution est proportionnel à la racine carré de n. Les deux algorithmes Rho de Pollard semble en effet suivre cette tendance.

Quant à Pollard 3, nous avons peut être identifié un facteur qui rend négligeable la taille de n, pour la plage de n étudiée. Il s'agirait du seuil de friabilité de ses facteurs premiers, qui donnerait, dans la majorité des cas, le nombre d'itérations, et donc le temps d'exécution pour factoriser n.

Références

[1] Pollard, J. M. (1975), "A Monte Carlo method for factorization", *BIT Numerical Mathematics*, 15 (3): 331–334

[2] Ronald, L. ; Shamir, A. ; Adleman, L. M. (1977) "Cryptographic communications system and method", *U.S. Patent Office* (US4405829A)

[3] Euler, L. (1763), "Theoremata arithmetica nova methodo demonstrata", *Novi commentarii academiae scientiarum imperialis Petropolitanae*, 74-104

[4] Bézout, É. (1779), *Théorie générale des équations algébriques*

[5] Floyd, R. W. (1967) "Non-deterministic Algorithms", *Journal of the ACM*, 636–644