

PROJET TUTORÉ  
Reconstituer le théorème chinois en présence  
d'erreurs

Mr LAVOIX John, Mr GRANERO Fabien

7 mai 2021

## Résumé

Nous sommes étudiants en troisième année de licence mathématique et informatique. Suite au cours d'algèbre des années précédentes et au cours d'arithmétique et cryptologie de ce semestre, nous avons décidé d'approfondir nos connaissances dans le domaine des restes chinois.

De plus, voulant tous les 2 poursuivre nos études en nous inscrivant au master de cryptologie et sécurité informatique, ce projet fut l'occasion de découvrir les prémisses des enseignements qui y sont dispensés.

Ce rapport est une synthèse de nos travaux durant ce semestre, sur la reconstitution du théorème des restes chinois en présence d'erreurs. Grâce au soutien de notre professeur Mr ZEMOR, qui nous a guidé tout au long de ce projet, nous avons découvert et maîtrisé plusieurs méthodes pour parvenir à notre objectif.

Pour ce faire, nous avons d'abord pris connaissance du théorème dans son ensemble. Puis dans un second temps, nous avons cherché et appris plusieurs facon de corriger le théorème des restes chinois lors de la présence d'erreurs.

# Table des matières

# Remerciements

Nous tenons tout d'abord à remercier monsieur CASTAGNOS Guillaume, d'avoir maintenu cette unité d'enseignement malgré la situation sanitaire actuelle. Nous remercions par la suite monsieur ZEMOR Gilles de nous avoir formé durant cette période compliquée, d'être resté à notre écoute, et de nous avoir suivi tout au long de ce projet.

Nous remercions également nos familles et nos proches, pour leur aide et leur soutien moral.

# Chapitre 1

## théorème des restes

### 1.1 théorème des restes chinois et son histoire

Les premières apparitions du théorème des restes chinois ont eu lieu au XIII<sup>e</sup> siècle dans un livre de mathématique chinois de Qin JUSHIO publié en 1247. Cependant, on avait déjà découvert ce théorème auparavant dans un livre de Sun ZI au III<sup>e</sup> siècle. Le théorème consiste en : On pose  $p_1, \dots, p_k$  des entiers premiers 2 à 2. Pour tout  $n_1, \dots, n_k$ , il existe un entier  $x$  tel que :

$$\begin{aligned}x &\equiv n_1 \pmod{p_1} \\x &\equiv n_2 \pmod{p_2} \\x &\equiv n_k \pmod{p_k}\end{aligned}$$

Il est difficile de trouver des applications immédiates au théorème des restes chinois. Pourtant il se révèle très pratique dans des comptages à grandes échelles pour calculer très rapidement en minimisant les possibilités d'erreurs ; ainsi qu'en cryptologie où la possibilité de faire des calculs sur de grands nombres avec des valeurs discrètes permet d'assurer une certaine sécurité si toutefois une partie des données fuitait. Cela s'applique particulièrement dans le domaine des cartes à puces (cartes bancaires, badges, cartes d'abonnement,...), où il y a une interaction avec un terminal pour transmettre un code. Il est très intéressant de délivrer l'information souhaitée par morceaux, surtout dans le cas où le terminal ferait l'objet d'une attaque. Dans ce cas, si l'ensemble des données récupérées par l'attaquant n'est pas trop important, alors il est impossible pour lui de les exploiter.

Nous allons dans un premier temps démontrer l'unicité de ce théorème. Tout d'abord, nous cherchons l'existence d'une solution :  
On a  $\forall k \in [1; y], x \equiv n_k \pmod{p_k}$

$\forall k \in [1; y]$ , on note  $P_k = \frac{P}{p_k}$  où  $P = p_1 \times \dots \times p_y$

On voit assez simplement que  $P_k$  et  $p_k$  sont premiers entre eux car tous les entiers  $p_i$  sont premiers entre eux 2 à 2. Donc  $P_k$  est inversible modulo  $p_k$ , car en faisant le théorème d'euclide, on va trouver  $u, v \in \mathbb{Z}$  tel que  $P_k \times u + p_k \times v = d$  avec  $d$  le PGCD. Or  $P_k$  et  $p_k$  sont premiers entre eux donc  $d = 1$  et on trouve facilement l'inverse de  $P_k$  modulo  $p_k$ .

On note alors  $u_k$  le nombre tel que  $u_k \times P_k + v_k \times p_k = 1$ , soit  $u_k \times P_k \equiv 1(p_k)$ . Soit  $x = \sum_{k=1}^y u_k \times P_k \times n_k$ . On pose  $i \in [1; y]$  et  $k \neq i$  alors  $P_i \equiv 0(mod p_k)$  car  $P_i = p_1 \times \dots \times p_k \times \dots \times p_y$ .

On a donc  $x \equiv u_i P_i a_i(mod p_i)$ , mais on sait que  $u_i P_i \equiv 1(mod p_i)$  d'où  $x \equiv a_i(mod p_i)$ . Sachant que cette équation est vraie pour tout  $i$ , on a x solution du système.

Par conséquent, nous avons prouvé l'existence d'une solution.

Ensuite, nous montrons l'unicité du système.

On a x une solution du système, posons y une autre solution du système. On a alors  $x - y \equiv 0(mod p_k)$ , soit  $p_k$  divise  $x - y$ . Sachant que les  $p_k$  sont premiers 2 à 2 entre eux, on a donc  $x - y \equiv 0(mod P)$ , donc P divise x-y, ou encore que  $x \equiv y(mod P)$ . Par cela, nous avons montré l'unicité modulo P.

Pour illustrer ce théorème, nous allons donner un exemple, extrait du livre de Sun ZI qui a proposé une solution.

Choisissons des objets comme des bonbons : si on les répartit pour 3 enfants, il en reste 2, si on les répartit pour les 3 enfants et leurs parents (soit 5 personnes), il en reste 3. Enfin si on rajoute les 2 cousins (soit 7 personnes), il reste alors 2 bonbons. On a donc :

$$x \equiv 2(mod 3)$$

$$x \equiv 3(mod 5)$$

$$x \equiv 2(mod 7)$$

La question que l'on se pose à présent est : combien y a t'il de bonbons?

Grâce au théorème des restes chinois, on peut trouver la réponse. Nous avons en réalité plusieurs réponses, c'est-à-dire que tout les nombres congrus à x modulo N sont des bonnes réponses. Avant de répondre à ce problème, il faut tout d'abord examiner l'algorithme.

## 1.2 Notre algorithme

Notre fonction en Python du théorème des restes chinois étant un peu lourde, nous allons montrer l'algorithme ci-dessous (voir annexe 1). On obtient donc :

Soit  $p_i$  le  $i$ -ème terme de la liste des modulus, on note

$$P_i = \frac{p}{p_i} = p_1 p_2 \dots p_{i-1} p_{i+1} \dots p_k$$

on a donc  $P_i$  et  $p_i$  qui sont premiers entre eux.

Il faut alors faire l'algorithme d'Euclide étendu sur  $P_i$  et  $p_i$ , ce qui nous donne :  $1 = P_i u_i + p_i v_i$  où  $u_i, v_i \in \mathbb{Z}$

On pose donc  $e_i = u_i P_i$  avec  $e_i \equiv 1 \pmod{p_i}$  et  $e_i \equiv 0 \pmod{e_j}$  avec  $i \neq j$

On trouve alors une solution qui est  $x = \sum_{i=1}^k p_i e_i$ .

Nous pouvons à présent répondre à l'exemple précédent. Nous avons :

$$x \equiv 2 \pmod{3}$$

$$x \equiv 3 \pmod{5}$$

$$x \equiv 2 \pmod{7}$$

On obtient  $P_1 = 5 \times 7 = 35$ ,  $P_2 = 3 \times 7 = 21$ , et  $P_3 = 3 \times 5 = 15$

On fait l'algorithme d'Euclide étendu sur  $P_1$  et  $p_1$  qui donne  $-3 \times 23 + 2 \times 35 \times 1 = 1$ , par conséquent on trouve  $e_1 = 35$

Idem sur  $P_2$  et  $p_2$  qui donne  $21 \times 1 - 5 \times 4 = 1$ , donc  $e_2 = 21$

Enfin, on a  $15 \times 1 - 7 \times 2 = 1$  avec  $e_3 = 15$

Le résultat est  $x = 2 \times 35 + 3 \times 21 + 2 \times 15 = 233$ . Nous avons, comme dit précédemment, plusieurs résultats qui sont les entiers congrus à 233 modulo 105.

$$233 \equiv 23 \pmod{105}$$

Le résultat final est donc  $23 + 105k$ ,  $k \in \mathbb{Z}$ . Si on reprend notre problème, on a donc 23 bonbons.

Comme vous pouvez le voir, nous avons utilisé l'algorithme d'Euclide étendu. Celui-ci prend en paramètres  $a$  et  $b$ , deux entiers. Il renvoie  $d$  le PGCD de  $a$  et  $b$  et un couple  $(u, v) \in \mathbb{Z}$  tel que  $d = au + bv$ .

Pour notre utilisation, nous avons un peu modifié cet algorithme, car pour utiliser le théorème des restes chinois, les nombres sont premiers entre eux 2 à 2, donc  $d = 1$ . De plus, il nous fait gagner une étape car il nous renvoie l'inverse de  $a$  modulo  $b$ .

---

**Algorithm 1** algorithme d'euclide étendu

---

```
 $x_0 \leftarrow 1$ 
 $x_1 \leftarrow 0$ 
 $y_0 \leftarrow 0$ 
 $y_1 \leftarrow 1$ 
 $s \leftarrow 1$ 
 $d \leftarrow b$ 
while  $b \neq 0$  do
   $(q, r) \leftarrow \text{divmod}(a, b)$  /* q est le quotient et r le reste de la division
  euclidienne de a par b */
   $(a, b) \leftarrow (b, r)$ 
   $(x, y) \leftarrow (x_1, y_1)$ 
   $(r_1, y_1) \leftarrow (q \times s_1 + x_0, q \times y_1 + y_0)$ 
   $(x_0, y_0) \leftarrow (x, y)$ 
   $s \leftarrow -s$ 
end while
return  $s \times x_0 + ((1 - s) \div 2) \times d$ 
```

---

Maintenant que nous avons présenté les algorithmes principaux du théorème des restes chinois, voici quelques fonctions que nous pourrions qualifier de secondaires au premier abord, mais très utiles dans certains cas.

En premier, nous avons fait un algorithme s'intitulant générateur de cas, qui prend en argument 2 nombres et qui génère les listes des nombres premiers entre eux, et la liste des restes modulo les  $p_i$ . C'est en quelque sorte la fonction inverse du théorème des restes chinois.

---

**Algorithm 2** algorithme d'encodage

---

**Require:**  $n > k$

```
 $l_n = \text{nombre} - \text{facteur} - \text{premier}(n)$ 
 $l_k = []$ 
for  $i$  in  $\text{range}(\text{len}(\text{ln}))$  do
   $l_k = l_k + [k/\text{ln}(i)]$ 
end for
return  $(l_n, l_k)$ 
```

---

Par exemple notre algorithme d'encodage sur (30030, 61) renvoie les listes  $P = [2, 3, 5, 7, 11, 13]$  et la liste  $N = [1, 1, 1, 5, 6, 9, 10]$ , qui représente la liste des facteurs premiers qui composent 30030 et la représentation de 61 modulo les éléments de  $P$ .

Nous pouvons remarquer que dans la fonction au dessus, nous utilisons la fonction nombre facteur premier qui prend en paramètre un nombre  $n$  et renvoie la liste des ses facteurs premiers.



---

**Algorithm 3** nombre facteur premier

---

```
 $l = []$   
 $i = 2$   
while  $n \neq 1$  do  
   $c = 1$   
  while  $n \% i = 0$  do  
     $n = n / i$   
     $c = c \times i$   
  end while  
  if  $(n \times c) \% i = 0$  then  
     $l = l + [c]$   
  end if  
   $i = i + 1$   
end while  
return  $l$ 
```

---

Par exemple nombre facteur premier (210) renvoie la liste  $[2, 3, 5, 7]$  car  $210 = 2 \times 3 \times 5 \times 7$

## Chapitre 2

# Presentation du problème

### 2.1 Le problème et ses restrictions

Le théorème des restes chinois induit une bijection :

$$\begin{aligned} [0, N-1] &\rightarrow \mathbb{Z}/p_1\mathbb{Z} \times \dots \times \mathbb{Z}/p_i\mathbb{Z} \\ x &\rightarrow (x_1, \dots, x_i) \end{aligned}$$

Où  $x \equiv x_i \pmod{p_i}$

Malheureusement nous ne pouvons encoder l'ensemble  $[0, N-1]$  dans son entièreté et être capable de corriger et faire face aux erreurs.

A propos du  $N$  à choisir ce qui nous intéresse dans le  $N$  choisi c'est un  $N$  entier produit de plusieurs nombres premiers.

Car plus il y a de nombres premiers qui composent  $N$  plus le système des restes chinois engendré par ce  $N$  contient d'équations.

Tout les nombres de 0 à  $N-1$  ne peuvent pas être "encodés" si l'on veut corriger des erreurs.

Il faut déterminer une borne  $B$  comprise entre 0 et  $N-1$  qui nous servira de "zone" d'encodage.

Pour déterminer cette borne on a 2 manières :

la première est de choisir  $N$  et de trouver  $B$  en se restreignant à une sous partie du système, le surplus servant de "donnée de parité" ;

la deuxième est de choisir  $B$  l'ensemble des entiers que l'ont veut encoder et rajouter des informations donc des équations supplémentaires au système engendré par  $B$ , donc multiplier  $B$  par d'autres nombres premiers.

## 2.2 Comment déterminer la borne

Maintenant comment déterminer B de façon calculatoire ?

Soit  $N = p_1 \times \dots \times p_n$  On pose  $x = p_1 \times p_2 \times \dots \times p_{n-2t}$  où t est le nombre d'erreurs.

On note la représentation de x par le théorème des restes chinois  $m = (0\dots 0a_1a_2\dots a_t\dots a_{2t})$

On reçoit le message  $r = (0\dots 0a_1\dots a_t0\dots 0)$

Nous obtenons 2 possibilités d'erreurs, soit :

-  $(0\dots 0a_1\dots a_t0\dots 0)$  avec des erreurs sur le t-uplet de  $a_1$  à  $a_t$ , qui correspond à l'encodage de 0.

-  $(0\dots 0a_1\dots a_t a_{t+1}\dots a_{2t})$  avec des erreurs sur le t-uplet de  $a_{t+1}$  à  $a_{2t}$ , qui correspond à l'encodage de x.

Donc la borne est x pour ne pas laisser de doute au moment de la correction.

Pour illustrer cette détermination de borne, voici un petit exemple :

Posons  $N = 2 \times 3 \times 5 \times 7 \times 11 \times 13 \times 17 = 510510$  et  $t = 2$ .

On trouve  $x = 2 \times 3 \times 5 = 30$ . Donc pour 2 erreurs, la borne de N est 30.

Dans les algorithmes que vous verrez dans la suite de notre travail, vous pourrez trouver nos calculs de borne. Cependant, ces calculs sont propres à chaque algorithme, c'est-à-dire que la borne pour un même nombre et un même nombre d'erreurs peut varier en fonction des algorithmes utilisés.

Cela signifie qu'après cette borne, la correction d'erreurs marche parfois, ce n'est donc plus une valeur sûre à cent pour cent, alors qu'avant cette borne, oui.

La question que nous sommes à présent en droit de nous poser est, est-ce qu'il existe une borne telle que l'on ne puisse corriger aucun nombre supérieur à cette borne ? Et la réponse est oui. Partons d'un n-uplet de départ  $x = (x_1, x_2, \dots, x_n)$ . On pose également, dans le n-uplet reçu, y le résultat possible avec t erreurs. Avec bien sûr, le cardinal des n-uplets reçu qui est inférieur à la sommes des  $p_i$ . Dans les n-uplets reçus, il y a un nombre D de résultats possibles. Nous cherchons donc à calculer D, qui vaut :

$$D = \sum_{I \subset [n], |I|=t} \prod_{i \in I} p_i - 1 \simeq \binom{n}{t} p^t \text{ avec } [n] \text{ les entiers de } 1 \text{ à } n \text{ et } p = p_n$$

Par exemple si  $x = (x_1, x_2, x_3, x_4)$  et  $t = 2$  alors  $D = (p_1 - 1)(p_2 - 1) + (p_3 - 1)(p_4 - 1) + (p_1 - 1)(p_3 - 1) + (p_1 - 1)(p_4 - 1) + (p_2 - 1)(p_3 - 1) + (p_2 - 1)(p_4 - 1)$

On a donc  $|recu| \simeq |B| \times D \leq \sum_{i=1}^n p_i$

Donc la borne absolue vaut :  $|B| \leq \frac{\sum_{i=1}^n p_i}{D}$ .

Idem que précédemment, faisons un petit exemple.

Posons  $N = 2 \times 3 \times 5 \times 7 \times 11 \times 13$  et  $t = 2$ , alors  $D \simeq \binom{6}{2} \times 13^2 = 2535$   
 $B \leq \frac{2 \times 3 \times 5 \times 7 \times 11 \times 13}{2535} = \frac{510510}{2535} \simeq 201$ .

## Chapitre 3

# Brute force

Il existe plusieurs algorithmes de brute force qui permettent de résoudre les erreurs tel que essayer tous les cas possibles. Mais ces algorithmes étant trop lourds à présenter dans ce dossier, nous avons décidé de les inclure en annexes. Nous avons ainsi choisi, pour la suite de notre travail, d'utiliser des algorithmes compatibles avec notre support.

### 3.1 Notre premier algorithme version naïve

Tout d'abord, nous avons fait une fonction qui nous montre s'il y a une erreur dans nos listes, car avant de vouloir trouver les erreurs, il faut savoir s'il y en a une (voir annexe 2). Nous avons donc fait un algorithme qui permet d'enlever alternativement un élément des listes. Autrement dit, on retire le premier élément des deux listes, puis on ne retire que le deuxième élément, puis que le troisième et ainsi de suite. Nous appliquons alors le théorème des restes chinois à chaque itération. Puis on met le résultat dans une liste auxiliaire. S'il n'y a pas d'erreur, lorsqu'on retourne cette liste auxiliaire, on devrait avoir la même valeur partout, en revanche si les valeurs de la liste sont différentes, nous avons une erreur.

Soit  $P = [p_1, p_2, \dots, p_k]$  et  $N = [n_1, n_2, \dots, n_k]$

On note  $P_i = [p_1, \dots, p_i - 1, p_i + 1, p_k]$  et  $N_i = [n_1, \dots, n_i - 1, n_i + 1, \dots, n_k]$

On note  $x_i$  le résultat théorème des restes chinois appliqué à  $P_i$  et  $N_i$

Puis on retourne  $L = [x_1, \dots, x_k]$

Si  $x_1 = x_2 = \dots = x_k$  alors il n'y a pas d'erreur.

Voici un exemple pour bien comprendre cette fonction.

On pose  $P=[2,3,5,7,11,13]$  et  $N=[1, 1, 1, 5, 6, 9]$

Si on applique le théorème des restes chinois à ces deux listes, on trouve 510510. Pour vérifier qu'il n'y a pas d'erreur, on applique le théorème à  $P1=[3,5,7,11,13]$  et  $N1=[1, 1, 5, 6, 9]$  puis à  $P2=[2,5,7,11,13]$  et  $N2=[1, 1, 5,$

6, 9] jusqu'à  $P_6=[2,3,5,7,11]$  et  $N_6=[1, 1, 1, 5, 6]$

Comme il n'y a pas d'erreur, la fonction va afficher [61, 61, 61, 61, 61, 61]

À présent, nous insérons une erreur dans les listes, on a  $P=[2,3,5,7,11,13]$  et  $N=[1,1,2,5,6,9]$ . En appliquant la fonction, il sera retourné une liste [6067, 6067, 61, 1777, 607, 1447], où l'on remarque clairement que chaque élément est différent. Donc nous avons au moins une erreur. Dans cette fonction, on parcourt  $n$  fois la liste de  $n$  éléments donc nous sommes en  $O(n^2)$ . De plus, on fait  $n$  fois l'algo des restes chinois donc on est en  $O(n^2)$ . Cette algorithmme est donc en  $O(n^4)$  avec  $n$  la taille de la liste. Cette fonction a également une borne qui est  $\prod_{i=1}^{n-1} p_i$ .

Pour créer notre première fonction de correction, nous nous sommes inspirés de la fonction ci-dessus (voir annexe 3). On s'est dit que si on enlevait 2 éléments des listes, nous pourrions trouver l'erreur. En effet, on a vu que si on enlevait 1 élément des listes, le théorème des restes chinois marchait encore, et si on enlève 2 éléments, il marche encore. Donc, en enlevant le premier élément et tous les autres chacun leur tour, et en appliquant le théorème à chaque fois, on retourne une liste. Puis, on réitère en enlevant le deuxième élément, et les autres chacun leur tour, en re-appliquant les restes chinois à chaque fois, en mettant les résultats dans une nouvelle liste. Ainsi de suite jusqu'au dernier élément. Cette fonction retourne une liste de liste. Donc la liste où se trouve le même nombre partout, signifie que l'erreur est à la même position. Faisons plutôt un exemple.

Soit  $P = [p_1, p_2, \dots, p_k]$  et  $N = [n_1, n_2, \dots, n_k]$

On note  $P_{i,j} = [p_1, \dots, p_i - 1, p_i + 1, \dots, p_j - 1, p_j + 1, \dots, p_k]$  et  $N_{i,j} = [n_1, \dots, n_i - 1, n_i + 1, \dots, n_j - 1, n_j + 1, \dots, n_k]$

On note  $x_{i,j}$  le résultat du théorème des restes chinois appliqués à  $P_{i,j}$  et  $N_{i,j}$

Et on retourne  $L = [[x_{1,1}, x_{1,2}, \dots, x_{1,k}], [x_{2,1}, x_{2,2}, \dots, x_{2,k}], \dots, [x_{k,1}, x_{k,2}, \dots, x_{k,k}]]$

On a retourné une liste de liste. Si la  $k$ -ième sous-liste retourne toujours le même nombre alors cela signifie qu'il y a une erreur sur le  $k$ -ième élément.

Si on récupère l'exemple du paragraphe précédent, où l'on avait découvert une erreur,

on a :  $P = [2, 3, 5, 7, 11, 13]$  et  $N = [1, 1, 2, 5, 6, 9]$

On applique notre algorithmme et il retourne [[6067, 1062, 61, 1777, 607, 292], [1062, 6067, 61, 347, 607, 677], [61, 61, 61, 61, 61, 61], [1777, 347, 61, 1777, 217, 127], [607, 607, 61, 217, 607, 187], [292, 677, 61, 127, 187, 1447]]

La première liste retournée ne renvoie pas le même résultat donc l'erreur n'est pas sur la première équation. Elle n'est pas non plus sur la deuxième équation. Cependant, on peut voir sur la troisième liste que le résultat est toujours le même; c'est à dire qu'à chaque fois, on a fait le théorème des restes chinois en enlevant la troisième équation et une autre, et on a trouvé le même résultat. Donc nous avons une erreur sur la troisième équation.

Voici un exemple d'échec de notre algorithmme pour des valeurs prises en de-

hors de la borne.  $N = [0, 2, 0, 5, 9, 11]$  est la liste représentant 2000 modulo les  $p_i$ . Posons une erreur sur  $N$ , tel que  $N = [0, 2, 0, 6, 9, 11]$  : l'algorithme renvoie  $[[12725, 2715, 713, 2000, 440, 20], [2715, 7720, 1714, 570, 440, 20], [713, 1714, 3716, 284, 440, 20], [2000, 570, 284, 2000, 50, 20], [440, 440, 440, 50, 440, 20], [20, 20, 20, 20, 20, 20]]$ . Comme on l'observe, l'algorithme trouve parfois 2 000 mais ne trouve pas l'erreur, ou du moins, il en trouve une mauvaise.

Cette fonction est très performante comme vous avez pu le voir sur les exemples, cependant il y a quelques défauts. Premièrement, cette fonction ne marche qu'avec une seule erreur, on peut facilement changer la fonction pour trouver n erreur juste en rajoutant des boucles FOR, mais cela rendrait l'algorithme encore plus lourd et aurait une borne très faible. Deuxièmement, nous avons un problème de borne. Pour trouver la borne, il suffit de multiplier par les 2 derniers nombres premiers de la liste.

Par exemple pour le nombre 30030, la borne est  $30030/(11 \times 13)$  soit 210, ce qui est très peu.

Enfin, nous avons un problème de complexité soit  $O(n^3)$  et à chaque fois on fait le théorème des restes chinois qui est en  $O(n^2)$ , ce qui donne un algorithme en  $O(n^5)$ .

## 3.2 brute force de hamming

Notre deuxième algorithme brut force consiste à utiliser la distance de Hamming. Soit  $x = x_1x_2...x_n$  et  $y = y_1y_2...y_n$ , la distance de Hamming est  $d_H(x, y) = \text{Card}\{i | x_i \neq y_i\}$ . Par exemple la distance de Hamming de  $x = 1, 2, 3, 4$  et  $y = 1, 2, 5, 6$  est 2

Avant de voir cette algorithme, nous devons expliquer la distance minimale. Soit  $G$  un groupe de  $n$ -uplet appartenant à  $A^n$ . La distance minimale de  $G$  est la plus petite distance de Hamming sur tout les éléments de  $G$ , soit  $d(G) = \min\{d_h(x, y) | x \neq y, x, y \in G\}$ . D'après Hamming, si la distance minimale est égal à  $d$ , alors on peut détecter  $d-1$  erreurs et corriger  $\frac{d-1}{2}$  erreurs.

Dans cette algorithme nous utilisons la distance de hamming qui prend en entrée 2 listes de nombres nommés  $A1$  et  $A2$ , et retourne la distance de Hamming. Du coup, il nous a fallu le coder.

---

### Algorithm 4 distance de hamming

---

```

 $n = \min(\text{len}(A1), \text{len}(A2))$ 
 $m = \max(\text{len}(A1), \text{len}(A2))$ 
 $cpt = m - n$ 
for  $i$  in  $\text{range}(n)$  do
    if  $A1[i] \neq A2[i]$  then
         $cpt = cpt + 1$ 
    end if
end for
return  $cpt$ 

```

---

Pour ce nouvel algorithme (voir annexe 4), nous allons calculer la distance de Hamming entre le uplet à corriger et tout les éléments de 0 à la borne. Nous sélectionnons tout les éléments de distance de Hamming strictement inférieur à 3 pour pouvoir détecter une erreur.

Pour choisir la borne de cette algorithme pour 1 erreur, on retire les 3 plus grandes équations, c'est à dire si  $P = [p_1, ..., p_k]$  alors la borne est  $\prod_{i=1}^{k-3} p_i$ . Pour  $e$  erreurs, on retire les  $3e$  dernières équations.

Faisons un exemple pour illustrer cette algorithme :

Prenons les listes  $P = [2, 3, 5, 7, 11, 13, 17]$  et  $N = [1, 0, 0, 5, 9, 10, 7]$  pour nos exemples.

Détaillons, pour mieux le comprendre, notre algorithme sur  $N = [1, 0, 0, 5, 7, 10, 7]$  c'est à dire que nous mettons une erreur sur le cinquième termes. Dans un premier temps, l'algorithme va faire la distance de Hamming entre la liste  $N$  et le nombre 0 qui s'écrit  $L_0 = [0, 0, 0, 0, 0, 0, 0]$  car le reste dans division euclidienne des  $p_i$  par 0 est toujours 0. La distance de Hamming de ces 2 listes est 5, car 2 éléments sur les 7 sont identiques entre les 2 listes, donc on

ne prete pas attention à ce cas.

On fait la distance de hamming entre le liste N et 1 qui s'écrit  $L_1 = [1, 1, 1, 1, 1, 1, 1]$  car le reste dans la division euclidienne des  $p_i$  par 1 est toujours 1. La distance de Hamming de ces 2 listes est 6 donc on ne prete pas attention. On fais la distance de Hamming entre la liste N et 2 qui s'écrit  $L_2 = [0, 2, 2, 2, 2, 2, 2]$ . La distance de Hamming de ces 2 listes est 7 donc on ne prete pas attention.

Il continue ses recherches sur les entiers 3, 4, et ainsi de suite. L'algorithme à découvert une liste qui avait une distance de Hamming inférieure à 3. Cette liste est la liste  $L = [1, 0, 0, 5, 9, 10, 7]$  qui correspond au nombre 75. Donc 75 est un candidat possible à la correspondance de cette uplet. l'algorithme va continuer ces tests sur tout les nombres jusqu'à notre borne moins 1 qui ici vaut  $2 \times 3 \times 5 \times 7 - 1 = 209$ .

De même que pour l'algorithme précédent, voici le même exemple.

Rapelons les faits,  $N = [0, 2, 0, 5, 9, 11]$ , et on implémente une erreur qui est :  $N = [0, 2, 0, 6, 9, 11]$ . L'algorithme retourne [20], c'est une nouvelle fois une erreur.

La complexité de cette algorithme se calcul assez rapidement. Nous avons une boucle while, on fait donc  $b$  fois la distance de Hamming. La distance de Hamming est donc  $O(r)$  où  $r$  est la taille du système. On a donc cette algorithme qui est en  $O(b \times r)$ .



## Chapitre 4

# Les fractions continues

### 4.1 Définition

Comme vu précédemment la méthode brute force est efficace sur des petits cas de correction mais sur de cas plus grand la complexité augmente considérablement et il est très difficile d'obtenir des résultats en un temps raisonnable.

Il faut donc trouver un moyen plus rapide et optimal. Donc, comprenons comment intervient l'erreur sur le n-uplet du système d'équation.

Soit  $m$  le message envoyé et  $y$  le message reçu avec l'erreur. Si l'on fait la différence de  $m-y$  on obtient  $e$  l'erreur. Maintenant cette erreur se représente par un uplet en supposant, par soucis de représentation, que les erreurs se trouve au début de l'uplet associé  $e = [e_1, e_2, e_3, \dots, e_k, 0, \dots, 0]$ . Cette erreur est donc un multiple de tout les  $p_i$  nombres premiers pour  $i > k$  et  $k$  reste non nul sur les positions erronés.

Donc  $y = m + e$ . Si l'on divise  $y$  par  $N$  notre entier on a  $\frac{y}{N} = \frac{m}{N} + \frac{e}{N}$ .

Mais  $\frac{e}{N}$  peut se simplifier car hormis les positions erronés,  $e$  est un multiple de tout les autres  $p_i$  qui composent  $N$

Ainsi on a  $\frac{y}{N} = \frac{m}{N} + \frac{e}{N} = \frac{m}{N} + \frac{e_1 * e_2}{p_1 * p_2} < \frac{B}{N}$   $B$  étant la borne choisie.

Finalement  $\frac{m}{N} = \frac{y}{N} - \frac{e_1 * e_2}{p_1 * p_2}$

Or souvenons-nous que si  $z$  est un réel positif et que la fraction rationnelle  $\frac{p}{q}$  vérifie  $\left| z - \frac{p}{q} \right| < \frac{1}{2q^2}$ . Alors  $\frac{p}{q}$  est une fraction réduite de  $z$ .

Appliquer à notre problème chaque réduite de cette fraction liste au dénominateur des candidats potentiels pour les positions erronés.

Lors de l'écriture des fractions continues en python, nous avons du faire appel à des fonctions secondaires (voir annexe 5). Tout d'abord, il nous a fallu une petite fonction qui calcul la borne maximal pour cette méthode.

---

**Algorithm 5** borne maximale pour les fractions continues

---

```
borne = 1
for i in range(len(N)-1,len(N)-t-1,-1) do
    borne = borne × N[i]
end for
return borne
```

---

Dans un second temps, nous avons utilisé la fonction réduite qui prend en paramètres les 3 entiers  $k$ ,  $n$  qui sont le numérateur et le dénominateur de la fraction d'origine et  $p$  l'ordre de la réduite de  $\frac{k}{n}$  et renvoie  $l$  la liste des  $a_i$  qui compose la réduite.

---

**Algorithm 6** fraction reduite

---

```
a = k
b = n
q = 1
r = 1
l = []
n = 0
while n < p do
    q = a//b
    r = a%b
    l = l + [q]
    a = b
    b = r
    n = n + 1
    if r = 0 then
        return l
    end if
end while
return l
```

---

Et enfin, nous retrouvons la fonction liste-int-réduite qui prend en paramètre  $l$  la liste des  $a_i$  la suite de polynôme qui compose la réduite, retourné par la fonction d'avant et qui renvoie 2 entiers le numérateur et le dénominateur de la réduite reconstitué

---

**Algorithm 7** liste entier réduite

---

```

 $h_2 = 0$ 
 $h_1 = 1$ 
 $k_1 = 0$ 
 $k_2 = 1$ 
 $hp = 0$ 
 $kp = 0$ 
for  $i$  in range(len( $l$ )) do
     $hp = l[i] \times h_1 + h_2$ 
     $kp = l[i] \times k_1 + k_2$ 
     $h_2 = h_1$ 
     $h_1 = hp$ 
     $k_2 = k_1$ 
     $k_1 = kp$ 
end for
return  $hp, kp$ 
```

---

Maintenant parlons de la complexité de cette méthode. On parcourt la liste des derniers nombres premiers du système, pour calculer une borne à ne pas dépasser au dénominateur de la fraction cela réduit drastiquement les itérations à faire. Une fois cette borne déterminée on calcule les réduites de  $\frac{y}{n}$  sans que le dénominateur qui sert d'indication sur la position des erreurs ne dépasse la borne. A chaque tour de boucle on calcule une réduite qui coûte l'ordre de la réduite fois le coût d'une division euclidienne, l'algorithme permettant de calculer une réduite étant une version modifiée de l'algorithme d'Euclide arrêté à l'ordre de la fraction réduite voulue. Donc finalement pour calculer toutes les réduites qui mettent en évidence les potentiels positions erronées le coût est relativement faible car dépend de la taille du système linéairement donc relativement faible et rapide. Coût que l'on a constaté grâce à nos expériences et tests.

## 4.2 Résumé d'expériences et test

Au début de nos tests nous avons pratiqué avec des petits cas de 6 à 7 nombre premier et cette méthode ne se révéla pas très efficace. Du a fait que le système étudié est trop petit, la méthode des fractions continues échoue par manque d'équation. Mais pour tester des cas avec plus d'erreurs nous avons eu besoin de système plus grand, nous avons donc directement étendu notre système à 51 nombres premiers. Nous avons pris la borne  $B = 10^{29}$  ordre de grandeur de la borne théorique pour corriger 15 erreurs. Dans ce cas et grâce à la méthode des fractions continue nous avons été en capacité de corriger jusqu'à 7 erreur.

Voici quelque exemple :

Pour le premier exemple, nous avons choisi de travailler avec les 20 premiers nombre premier, et  $[1, 2, 1, 1, 0, 8, 1, 18, 4, 23, 10, 12, 6, 0, 9, 51, 50, 26, 61, 61]$  le  $n$ -uplet avec 2 erreurs. L'algorithme retourne  $[[2], [3, 43]]$ . On reconstitue le théorème en enlevant, dans un premier temps l'équation du modulo 3. Et dans un autre temps, nous avons fait le théorème en retirant les équations modulo 3 et 43. Nous avons deux valeurs retourné, suite à la reconstitution du théorème, qui sont 2162561357080315373769686 pour la première et 86842888231 pour le second. Le bon résultat étant le plus petit, cela signifie que l'erreur est sur l'équation modulo 3 et modulo 43.

De plus, on pose pour les  $p_i$  les 50 premiers nombres premiers, et  $N = [0, 2, 0, 4, 7, 1, 16, 2, 22, 25, 26, 34, 18, 19, 30, 24, 24, 22, 52, 56, 4, 15, 54, 68, 89, 58, 39, 15, 54, 11, 112, 73, 134, 126, 102, 142, 142, 92, 159, 48, 119, 33, 16, 77, 28, 29, 67, 89, 86, 163, 150]$  le  $n$ -uplet reçu, c'est à dire avec des erreurs(7). Les erreurs sont sur les positions correspondant aux nombres premiers de la liste suivante  $[7, 79, 97, 107, 127, 157, 223]$ . L'algorithme retourne  $[[5], [2, 7], [3, 11], [7, 79, 97, 107, 127, 157, 223]]$ . Comme nous pouvons l'observer, l'algorithme à trouvé la bonne réponse. Cependant il propose d'autres cas. Par le même biais que l'exemple précédent, nous trouver que la bonne solution est le 7-uplets.

# Conclusion

Finalement on peut distinguer 2 cas. Premièrement si l'on doit corriger quelques erreurs sur un petit système les méthodes brute force évoquées précédemment sont à préconiser malgré leur coût très élevé et dans un second cas sur des systèmes plus grands de taille 20 ou plus avec plus d'erreurs alors la méthode des fractions continues se révèle beaucoup plus efficace voir nécessaire pour espérer avoir un résultat comparé à la méthode brute force. Dans des cas concrets appliqués à la vie de tous les jours on se retrouve face à des grands cas comme dans le domaine des cartes à puces etc .. alors la méthode des fractions continues est à choisir .

# Histoire du théorème

Comme nous l'avons vu précédemment, les premières traces connues de ce théorème remontent environ au III<sup>e</sup> siècle, où un mathématicien nommé maître SUN exposa un problème dans son "classique". Maître Sun Zi ou Sun Tzu est un mathématicien et astronome chinois, ayant vécu entre le III<sup>e</sup> siècle et le V<sup>e</sup> siècle. C'est d'ailleurs le besoin d'éditer des calendriers astronomiques précis, qui poussa Sun Zi à énoncer le théorème des restes chinois dans son livre *Sun Tzu Suan Ching* ("Classique mathématique de Sunzi") pour pouvoir calculer les points de convergence des périodes de révolution des astres. Le problème de Sun Zi est celui énoncé précédemment où le résultat était 23. Ce problème a ensuite disparu pendant plusieurs siècles car on ne retrouve aucune trace, ni de ce problème, ni de problème ressemblant. Il faut attendre le XIII<sup>e</sup> siècle pour retrouver des traces du théorème. Comme par exemple dans le *shushu jiuzhang* de Qin Jiushao, où il sera donné un algorithme général de résolution. C'est une méthode appelée le *dayan* qui généralise le procédé de maître SUN. Le problème de Sunzi apparaît également au début du XIII<sup>e</sup> en Europe, où Leonard de Pise, dans *Liber abbaci*, écrira pratiquement au mot près, les mêmes solutions que dans le Sunzi. À partir de son arrivée en Occident, le problème de résolution des congruences simultanées commence à trouver son heure de gloire, où le grand mathématicien Gauss, écrira la définition des congruences et leur constitution en arithmétique moderne.

La question que l'on se pose, est pourquoi était utilisé ce théorème ? D'après les écrits et quelque hypothèse, il aurait servi en astronomie, où les chinois, qui adoraient ce domaine, souhaitaient calculer les correspondances des calendriers. Par exemple, Qin Jiushao cherchait dans combien de temps, se sera le quatrième jour de l'année, le huitième jour du mois lunaire et le dernier jour du cycle sexagénère.

Aujourd'hui, le théorème est utilisé dans d'autres domaines tel qu'en arithmétique et en algèbre, notamment sous sa forme générale dans l'arithmétique des corps, que se soit au cours de démonstrations théoriques aussi bien que dans des cas pratiques.

Dans le domaine de l'algorithmique, il est par exemple utilisé dans l'algorithme RSA en cryptographie, et il intervient aussi dans l'algorithme de Silver-Pohlig-Hellman pour le calcul du logarithme discret. Il intervient dans

l'algorithme de test de primalité de Agrawal et Biswas, développé en 1999[13].

Il permet de représenter de grands nombres entiers comme  $n$ -uplets de restes de divisions euclidiennes. Sous cette forme, des opérations comme l'addition ou la multiplication peuvent se faire en parallèle en temps constant (pas de propagation de retenue). Par contre, la comparaison ou la division ne sont pas triviales

# Annexes

## Annexe 1

---

**Algorithm 8** algorithme du théorème des restes chinois

---

**Require:**  $\text{len}(P) = \text{len}(N)$

$b = 0$

**for**  $i$  in  $\text{range}(\text{len}(N))$  **do**

$p_i = 1$

$pi = P[i]$

$k = 1$

**for**  $j$  in  $\text{range}(\text{len}(N))$  **do**

**if**  $j \neq i$  **then**

$p_i = p_i \times P[j]$

**end if**

**end for**

**while**  $k \times p_i \neq pi \times y + 1$  **do**

$k = k + 1$

**end while**

$b = b + k \times p_i \times N[i]$

**end for**

**return**  $b$

---



Annexe 2

---

**Algorithm 9** algorithme pour savoir s'il y a une erreur

---

**Require:**  $\text{len}(P) = \text{len}(N)$

$L = []$

**for**  $i$  in  $\text{range}(\text{len}(N))$  **do**

$N1 = []$

$P1 = []$

**for**  $j$  in  $\text{range}(\text{len}(N))$  **do**

**if**  $i \neq j$  **then**

$N1 = N1 + [N[j]]$

$P1 = P1 + [P[j]]$

**end if**

**end for**

$l = \text{reste} - \text{chinois}(N1, P1)$

$L = L + [l]$

**end for**

**return**  $L$

---

Annexe 3

---

**Algorithm 10** brute force correction 1 erreur

---

**Require:**  $\text{len}(P) = \text{len}(N)$   
 $L1 = []$   
**for**  $i$  in  $\text{range}(\text{len}(N))$  **do**  
     $L = []$   
    **for**  $j$  in  $\text{range}(\text{len}(N))$  **do**  
         $N_i = []$   
         $P_i = []$   
        **for**  $k$  in  $\text{range}(\text{len}(N))$  **do**  
            **if**  $k \neq i$  and  $k \neq j$  **then**  
                 $N_i = N_i + [N[k]]$   
                 $P_i = P_i + [P[k]]$   
            **end if**  
        **end for**  
         $l = \text{reste} - \text{chinois}(N_i, P_i)$   
         $L = L + [l]$   
    **end for**  
     $L1 = L1 + [L]$   
**end for**  
**return**  $L1$

---

#### Annexe 4

Cette algorithmme prend en paramètre la liste des modulus, la liste des restes, et le nombre d'erreur. Et retourne la liste des candidats correspondant au uplet fourni.

---

**Algorithm 11** brute force de hamming

---

```
n = len(N)
cpt = 0
N = 1
borne = 0
for i in range(n) do
    N = N × P[i]
    borne = borne + (P[i] − 1)
end for
borne = N / borne
(a, L − force) = generateur − de − cas(N, cpt)
l − candidat = []
while cpt ≤ borne do
    if distance − hamming(L − force, N) ≤ nb − erreur then
        l − candidat = l − candidat + [cpt]
    end if
    cpt = cpt + 1
    (a, L − force) = genrateur − de − cas(N, cpt)
end while
return l-candidat
```

---

Annexe 5 Cette fonction prend en paramètre la liste des  $p_i$  la liste des restes modulus les  $p_i$ , et le nombre d'erreur et retourne la liste des listes d'équations supposées fausses.

---

**Algorithm 12** fraction continue

---

```

k = reste - chinois(P, N)
p = 1
n = 1
L = []
for i in range(len(P)) do
    n = n × P[i]
end for
borne = max - borne - frac(N, nb - erreur)
b = 0
while b ≤ borne do
    L1 = fraction - reduite(k, n, p)
    (a, d) = list - int - reduite(L1)
    L2 = decompose - liste(b, N)
    if L2 ≠ −1 and L2 ≠ [] then
        L = L + [L2]
    end if
    p = p + 1
end while
return L

```

---

# bibliographie et sitographie

<https://cm2.ens.fr/content/le-problème-des-restes-chinois-questions-sur-ses-origi>  
<http://www.bibmath.net/dico/index.php?action=affiche&quoi=./c/chinois>.  
[html http://epsilon.2000.free.fr/Csup/resteschinois.pdf](http://epsilon.2000.free.fr/Csup/resteschinois.pdf) [http://math.univ-lyon1.fr/irem/IMG/pdf/Th\\_restes\\_chinois.pdf](http://math.univ-lyon1.fr/irem/IMG/pdf/Th_restes_chinois.pdf)

Kaoru Baba et Kiyosi Yabuuti, Une histoire des mathématiques chinoises, Belin.

Jean-Claude Martzloff, Histoire des mathématiques chinoises, p. 129

J.-C. Martzloff, Histoire des mathématiques chinoises, p. 296.

Ancienne, écrite par Arnaud Gazagnes et publiée par l'IREM de Reims en 2005

D. M. Mandelbaum, On a class of arithmetic codes and a decoding algorithm, IEEE Trans. on Information Theory 21(1) 8588, 1976.

Une révolution de la théorie des nombres GAUSS, RBA

$$\begin{aligned}
 V &= (n_1 \quad n_2 \quad \cdots \quad n_k) \\
 M &= \begin{pmatrix} e_1 & 0 & \cdots & 0 \\ 0 & e_2 & \cdots & 0 \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & 0 & e_k \end{pmatrix} \\
 V &= (2 \quad 3 \quad 2)
 \end{aligned}$$

$$M = \begin{pmatrix} 70 & 0 & 0 \\ 0 & 21 & 0 \\ 0 & 0 & 15 \end{pmatrix}$$