Complexité algébrique et cryptographie

Alexandre Guillemot

15 décembre 2022

Table des matières

Chapitre 1

Problèmes difficiles en théorie des nombres

1.1 Complexité et cryptographie

1.1.1 Introduction

Idée : mesurer la "difficulté" algorithmique d'un problème.

Définition 1.1.1. (Problème de décision) Un problème de décision est une collection d'instances qui sont des ensembles de données qui admettent exactement une des deux réponses "oui" ou "non".

Ex 1.1.1. 1. Problème SAT (Satisfaisabilité)

Instance : Une fonction à variables booléenne $F : \{0,1\}^n \to \{0,1\}$ construite avec les connecteurs logiques $\vee, \wedge, \neg$. Par exemple,

$$f(x_1, x_2, x_3, x_4) = (\neg(x_1 \wedge (\neg x_3))) \vee (x_1 \wedge x_2 \wedge (\neg x_1))$$

Question : existe-t-il $x_1, \cdots, x_n \in \{0,1\}$ temls que $F(x_1, \cdots, x_n) = 1$?

Algorithme : recherche exhaustive sur $(x_1, \cdots, x_n)$, la complexité est en $\mathcal{O}(2^n)$.

2. FBQ (Formes Booléennes Quantifiées)

Instance : Une formule booléenne avec quantificateur e.g. $\forall x_i \exists x_j \cdots F(x_1, \cdots, x_n)$ (F est une fonction booléenne comme dans SAT)

Question : Cette formule est-elle vraie ?

Algorithme : Recherche exhaustive ($\mathcal{O}(2^n)$).

3. Equations diophantiennes ($10^{\text{ème}}$ problème de Hilbert)

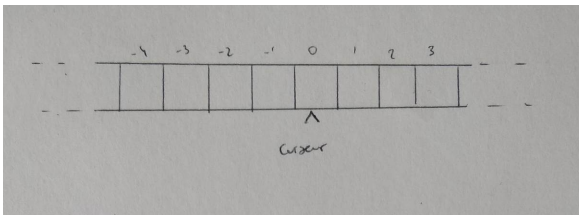
Instance : Une équation polynomiale à plusieurs inconnues et à coefficients entiers
Question : Cette équation admet-elle des solutions entières ?
Algorithme : Matyasevich, 1971 : il n'y a pas d'algo qui répond à cette question.

1.1.2 Calculabilité au sens de Turing

Turing : Cryptanalyse d'Enigma, construction de machines dédiées à la cryptanalyse d'Enigma, Machine de Turing.

Modèle de Turing

On dispose d'un ruban infini



Chaque case contient un symbôle (dans un alphabet fini Σ que l'on peut supposer être $\{0,1\}$, ou le symbôle blanc b). Le ruban v aêtre lu case par case par le curseur, la machine est à chaque instant dans un état $q_i \in Q$, où Q est l'ensemble fini des états possibles.

Définition 1.1.2. (Machine de Turing) Une opération élémentaire est entièrement déterminée par le symbôle lu par le curseur, et par l'état actuel q_i :

1. Le curseur remplace le symbôle par un élément de $\Sigma \cup \{b\}$
2. Le curseur de déplae soit d'une case vers la gauche, soit d'une case vers la droiten, soit reste sur place.
3. La machine passe de l'état q_i à l'état q_j.

Une machine de Turing est donc la donnée d'une fonction

$$M : (\Sigma \cup \{b\}) \times Q \to (\Sigma \cup \{b\}) \times \{-1, 0, 1\} \times Q$$

Définition 1.1.3. (Calcul déterministe) Le calcul déterministe d'une entrée x avec une machine de Turing M est la suite d'opération suivante :

1. La machine est commence par être dans l'état q_0
2. Le curseur est placé sur la case 1
3. x est écrite sur les cases $1, \cdots, n$ du ruban, les autres contenant b.

4. On applique itérativement M, le calcul se terine lorsque la machine atteint l'état final q_F. La sortie y est alors la donnée inscrite sur le ruban lorsque la machine termine.

Terminologie : L'ensemble des suites finies de symbôles de Σ est noté Σ^*. Un mot est un élément de Σ^*. Une fonction $f : \Sigma^* \to \Sigma^*$ est turing calculable s'il existe une machine de Turing M qui sur tout entrée $x \in \Sigma^*$ calcule $y = f(x)$.

1.1.3 Complexités en temps et classes P, NP

Définition 1.1.4. La longeuru d'un calcul sur une entrée $x \in \Sigma^*$ pour une machine de Turing M est le nombre $t_M(x)$ d'opération élémentaires qui composent le calcul. Ainsi on défini la complexité en temps d'une machine de Turing comme

$$\begin{array}{rccc} T_M : & \mathbb{N} & \to & \mathbb{N} \\ & n & \mapsto & \max_{\substack{x \in \Sigma^* \\ |x|=n}} \{t_M(x)\} \end{array}$$

Définition 1.1.5. Un algorithme polynomial $\mathcal{A}$ pour calculer f est une machine de Turing M qui calcule f et telle qu'il existe un polynôme p tel que $\forall n \in \mathbb{N}$, $T_M(n) \leq p(n)$. On appelle classe P l'ensemble des problèmes de décision admettant un algorithme polynomial.

Définition 1.1.6. On dit qu'un pb de décision est calculable par un algo non déterministe polynomial s'il existe une machien de Turing M et un polynome p tel que

1. La réponse est oui pour l'entrée x ssi il existe $y \in \Sigma^*$ (certificat) tel que M calcule 1 lorsqu'on met $x \in \Sigma^*$ dans les cases 1 à n et y dans les cases -1 a $-m$.
2. Pour tout x donnant la réponse 1, M calcule 1 et temps $\leq p(n)$

On appelle classe NP la classe des problèmes de décision admettant un algorithme non déterministe polynomial.

Rq 1.1.1. $P \subseteq NP$.

1.1.4 Problèmes NP-complets

Définition 1.1.7. On dit que le problème de décision p_1 se réduit au problème de décision p_2 s'il existe une fonction $\varphi : \Sigma^* \to \Sigma^*$ calculable en temps polynomial telle que la réponse à p_1 est oui pour l'entrée x si et seulement si la réponse à p_2 est oui pour l'entrée $\varphi(x)$.

Notation. On note $p_1 \ltimes p_2$.

Proposition 1.1.1. *$p_1 \in P$ et $p_1 \ltimes p_2 \Rightarrow p_1 \in P$*

Définition 1.1.8. Un problème Π est NP-complet ssi $\forall p \in NP$, $p \ltimes \Pi$.

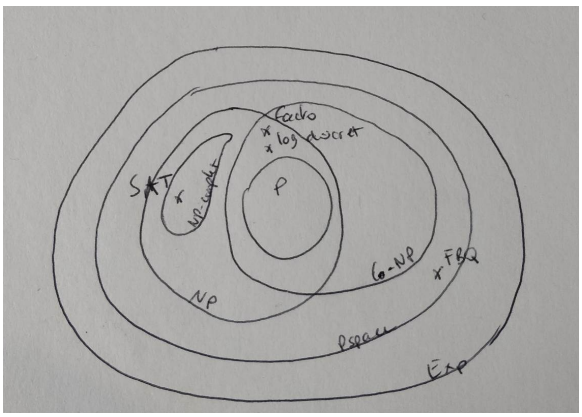
Théorème 1.1.1. *(Cook, 1971) SAT est NP-complet.*

Rq 1.1.2. Si $SAT \ltimes P$, alors P est NP-complet.

Ex 1.1.2.
1. SAT
2. 3-SAT
3. Circuit hamiltonien
4. 3-coloriabilité d'un graphe
5. TSP
6. Pb du sac à dos
7. Système de n équations quadratiques sur un $\mathbb{F}_2$.

On conjecture que $P \neq NP$. Astuce de Levin : Supposons que $P = NP$: alors on peut construire un algorithme polynomial pour résoudre SAT. Considérons les machines de Turing $M_1, M_2, \cdots$ qui prennent en entrée une instance de SAT : Alors on fait tourner les machines simultanément, en faisant tourner de une étape M_1, puis en faisant tourner de une étape M_2, puis M_1, puis M_3, M_2 et M_1, etc.

Résumé de la hiérarchie des complexités algorithmiques :



Rq 1.1.3. Co-$NP \cap NP$-complet $= \emptyset$.

1.2 Factorisation

1.2.1 Complexité

Problème de décision FACTM (problème des facteurs majorés)

- Instance : n entier, $M \leq n$.
- Question : Existe-t-iil un diviseur de n qui est $\leq M$.

Si on a un algo polynomial de factorisation, alors on peut résoudre FACTM en temps polynomial. Inversement, Supposons $\mathcal{A}$ algo polynomial pour FACTM. Comment factoriser n ? Soit p le plus petit facteur premier de n, alors

- On applique $\mathcal{A}(n, \sqrt{n})$. Si l'algorithme répond non, alors on termine et on répond non (car n est alors premier)
- Sinon, on applique $\mathcal{A}(n, \sqrt{n}/2)$. Si l'algo répond non, alors $p \in [\sqrt{n}/2, \sqrt{n}]$, et sinon $p \in [1, \sqrt{n}/2]$.
- On continue la dichotomie jusqu'à ce que la taille de l'intervalle obtenu soit plus petite que 1.

L'algorithme termine dès que $\sqrt{n}/2^k < 1$, où k est le nombre d'appels de $\mathcal{A}$. Ainsi il y a $k = \log_2(\sqrt{n})$ est donc de l'ordre de $\log n$. Une fois p trouvé, on recommence l'algo avec n/p. On va recommencer le nombre de facteurs premiers de n (comptés avec leur multiplicité). Mais

$$n = \prod_i p_i^{\alpha_i} \geq \prod_i 2^{\alpha_i} = 2^{\sum \alpha_i}$$

donc $\sum \alpha_i < \log_2 n$, et c'est aussi le nombre de facteurs premiers de n (comptés avec leur multiplicité). Au total, l'algorithme est polynomial.

1.2.2 Idée de Fermat

- L'idée naïve est d'essayer de diviser par les entiers successifs $\leq n$. C'est en $\mathcal{O}(n)$ donc exponentiel en la taille de l'entier.
- On peut aussi s'arrêter avant $\sqrt{n}$, mais l'algorithme reste exponentiel.
- On peut aussi diviser par les nombres premiers $\sqrt{n}$. D'après le théorème des nombres premiers (Hadamard, de la Vallée-Poussin), le cardinal des entiers premiers plus petits que x est aymptotiquement équivalent à $x/\ln x$. Ainsi l'algo est en $\mathcal{O}(\sqrt{n}/\log n)$, qui reste exponentiel en la taille de n.

Il suffit, pour factoriser n, de trouver x et y tels que $x^2 = y^2[n]$, avec $x \neq \pm y[n]$. En effetn on a alors $(x-y)(x+y) = 0[n]$ et ainsi $\gcd(x, x-y)$ est un facteur de n. Fermat prend comme valeurs de x $\lfloor\sqrt{n}\rfloor + 1, \lfloor\sqrt{n}\rfloor + 2$, et espère que $x^2 - n$ est un carré parfait y^2. SAUCISSE.

Ex 1.2.1. $n = 9167$, $\sqrt{n} = 95,7$, $96^2 = 49[n]$, et $49 = 7^2$, et alors $\gcd(9167, 96+7) = 103$, $\gcd(9167, 96-7) = 89$. On a bien $9167 = 103 \times 89$.

La complexité est de l'ordre de $\sqrt{n}$, c'est donc toujours exponentiel. Donnons un rafinement de la méthode : prenons $n = 849239$, $\sqrt{n} = 921,5$.

- $922^2 = 845 = 5 \times 13^2[n]$
- $933^2 = 2 \times 5^4 \times 17[n]$
- $937^2 = 2 \times 5 \times 13^2 \times 17[n]$

Et alors $(922 \times 933 \times 937)^2 = (2 \times 5^3 \times 13 \times 17)^2[n]$ et donc on a trouvé l'équation qu'on voulait, et après calcul des pgcd on obtiens que $1229 \times 691 = 849239$. On vient de décrire le crible quadratique de Pomerance.

Algorithmes

Décrivons le dans sa généralité : on veut factoriser n, pour cela

1. On se fixe une base de factorisation $B = \{-1, p_1, p_2, \cdots, p_h\}$.
2. On dit qu'un entier est friable (ou lisse/smooth) s'il n'a que des petits facteurs premiers. Précisément, il est B-friable si tous des facteurs premiers sont dans B.
3. On va dire que b est B-adapté si le représentant de $b^2[n]$ dans l'intervalle $[-n/2, n/2]$ est B-friable.

Etape 1 : Obtenir et stocker des entiers b_i qui sont B-adaptés. On note

$$b_i^2 = (-1)^{\varepsilon_i} p_1^{\alpha_{i,1}} \cdots p_h^{\alpha_{i,h}}[n] \tag{1.1}$$

Etape 2 : A chaque relation 1.1, on associe

$$u_i = (u_{i,0}, \cdots, u_{i,h}) \in \mathbb{F}_2^{h+1}$$

où $u_0 = \varepsilon_i[2]$, $u_{i,j} = \alpha_{i,j}[2]$ si $j \geq 1$.

Etape 3 : on a

$$\begin{aligned} \Rightarrow \prod_{i \in I} (b_i^2)^{\beta_i} &= \prod_{i \in I} \left((-1)^{\varepsilon_i} \prod_{j=1}^{h} p_j^{\alpha_{i,j}} \right)^{\beta_i} [n] \\ &= (-1)^{\sum_{i \in I} \beta_i \varepsilon_i} \times \prod_{j=1}^{h} p_j^{\sum_{i \in I} \beta_i \alpha_{i,j}} [n] \end{aligned}$$

donc si on trouve une combinaison linéaire des u_i qui est nulle

$$\sum_{i \in I} \beta_i u_i = 0$$

les exposants dans la dernière ligne du calcul sont pairs. On a donc $x^2 = y^2[n]$, avec

$$x := \prod_{i \in I} b_i^{\beta_i},\ y = \prod_{j=1}^{h} p_j^{\frac{1}{2}\sum \beta_i \alpha_{i,j}}$$

Complexité de l'algorithme

Etape 2 : Il faut $|I| \geq h + 2$.

Etape 1 : Pour évaluer la complexité de l'étape 1, on utilise

Théorème 1.2.1. *On pose*

$$\psi(x, T) = |\{n \leq x \mid n \text{ a tous se facteurs premiers } \leq T\}|$$

Pour $1 \leq T \leq x$, *on pose* $v := \frac{\ln x}{\ln T}$, *alors*

$$\frac{\psi(x, T)}{x} = v^{-v+o(1)}$$

On prend $B = \{-1, p_1, \cdots, p_h\}$, avec $p_1, \cdots, p_h$ les entiers premiers qui sont $\leq T = \exp\left(\frac{1}{2}\sqrt{\ln n \ln\ln n}\right)$. Alors

$$v = \frac{\ln\sqrt{n}}{\ln T} = \frac{\frac{1}{2}\ln n}{\frac{1}{2}\sqrt{\ln n \ln\ln n}} = \sqrt{\frac{\ln n}{\ln\ln n}}$$

donc $\ln v \simeq \frac{1}{2}\ln\ln n$. Ainsi le nombre de valeurs à essayer dans la première étape est $(h+2)v^v$. Et

$$v^v = \mathrm{e}^{v \ln v} = \mathrm{e}^{\frac{1}{2}\sqrt{\ln n \ln\ln n}} = T$$

Ainsi le nombre d'essais vaut $T(h+2)$, et $h + 2 \sim T/\ln T$.

Etape 3 : Finalement, la complexité de l'algorithme complet vaut

$$\frac{T^2}{\ln T} + (h+2)^3 \sim \left(\frac{T}{\ln T}\right)^3 = \mathrm{e}^{\frac{3}{2}\sqrt{\ln n \ln\ln n}}$$

On vient donc de décrire un algorithme sous-exponentiel.

Notation.

$$L_{\alpha,c}(z) = \mathrm{e}^{c(\ln Z)^{\alpha}(\ln\ln z)^{1-\alpha}}$$

- $\alpha = 0$: alors $L_{0,c}(z) = \mathrm{e}^{c\ln\ln z} = (\ln z)^c$ donc complexité polynomiale.

- $\alpha = 1$: $L_{1,c}(z) = \mathrm{e}^{c \ln z}$ donc complexité exponentielle.
- $0 \leq \alpha \leq 1$, alors $L_{\alpha,c}(z)$ est sous-exponentiel.

Dans le cas de l'algorithme que l'on vient de décrire, la complexité vaut $L_{\frac{1}{2},c}$. Actuellement, le meilleur algorithme pour des nombres types clés de RSA est GNFS (General Number Field Sieve) dont la complexité est $L_{\frac{1}{3},c}(n)$ avec $c = 1,92$ (algorithme du à H. Lenstra, A. Lenstra, Manasse, Pollard, 1990).

Rq 1.2.1. Si on veut $L_{\frac{1}{3},c}(n) \geq 2^{80}$, il faut prendre $|n| = 1024$ bits.

Rq 1.2.2. L'exposant 3 qui vient du pivot de gauss (3eme étape) peut être amélioré : le système à résoudre est un système creux :

$$b_i^2 = \prod_{j=1}^{h} p_j^{\alpha_{i,j}} [n]$$

Le nombre de facteurs premiers qui interviennent dans cette décomposition est de l'ordre $\mathcal{O}(\ln n)$. Ainsi les lignes du système à résoudre contiennent beacoup de zéros, et il existe un algorithme (Block-Lanczos) pour ce genre de système qui est en $\mathcal{O}(dh^2)$ où h est la dimension du système et d est le nombre macimal d'éléments non nuls dans chaque ligne.

1.3 Logarithme discret

1.3.1 Complexité

Problème du log discret : soit p un nombre premier, g un générateur de $\mathbb{Z}/p\mathbb{Z}^*$. À partir de $y = g^x[p]$, retrouver x ? On peut lui associer le problème de décision suivant :

Instance : p, g, y, t

Question : Le log discret de y par rapport à g est-il $\leq t$?

Si on a un algo A polynomial pour le problème de décision, alors de manière similaire au problème de factorisation (dichotomie), on peut calculer le log discret en temps polynomial.

1.3.2 Méthodes de calcul du log discret

- Méthode naïve : recherce exhaustive sur x, complexité en $\mathcal{O}(p)$ (donc exponentiel).
- Méthode baby step giant step : On regarde la division euclidienne de x par a où $a = \lfloor \sqrt{n} \rfloor$. Trouver x est équivalent à trouver q, r et poser $x = aq + r$. Maintenant

  $$y = g^x[p] \iff yg^{-aq} = g^r[p]$$

  On peut créer 2 tables de 0 à a indexées par q et r où on calcule yg^{-aq} et g^r, et on cherche une valeur commune. Si les tables sont triées, alors la recherche d'une valeur commune est en $\mathcal{O}(a)$.

1.3.3 Méthode du calcul d'indices

On cherche x tel que $g^x = y[p]$.

Etape 1 : On choisit $B = \{-1, p_1, \cdots, p_h\}$

- On choisit c_i aléatoire
- On calcule le représentant de $g^{c_i}[p]$ dans $[-p/2, p/2]$ et on espère que $g^{c_i}[p] = \prod_{j=0}^{h} p_j^{\alpha_{i,j}}$ $(*)$, dans ce cas on aura $c_i = \sum_{j=0}^{h} \alpha_{i,j} \log_g(p_j)[p_1]$.

1Etape 2 : Si on a obtenu $\geq h+1$ relations du type $(*)$, on pourra trouver les $\log_g(p_j)$ pour $0 \leq j \leq h$.

Etape 3 : On calcule $yg^e[p]$ où e est aléatoire. Avec une certaine probabilité, $yg^e = \prod_{j=0}^{h} p_j^{\beta_j}$, et alors

$$\log_g(y) = \left(\sum_{j=0}^{h} \beta_j \log_h(p_j) \right) - e$$

Par un argument similaire à l'analyse de complexité de l'algorithme de factorisation, on obtiens une complexité en $\mathcal{O}(\mathrm{e}^{1+o(1)\sqrt{\ln p \ln\ln p}})$, doit du $\mathcal{O}(L_{\frac{1}{2},1+o(1)}(p))$. Le meilleur algorithme est en $L_{\frac{1}{3},c}$ avec $c = 1,92$.

Chapitre 2

L'algorithme RSA en pratique

2.1 Rappels sur RSA

2.1.1 Définition

Histoire

1976 : Diffie Hellman, New Directions in Cryptography. 1977 : Merkle, "puzzle de Merkle". Rivest, Shamir, Adleman, RSA.

On fixe e impair ($e = 3, e = 17, e = 257$). On calcule des entiers p, q premiers distincts tels que $\gcd(e, (p-1)(q-1)) = 1$. On pose $n = pq$. Finalement, on calcule $d = e^{-1}[\varphi(n)]$. Clé publique : (n, e). Clé secrète : $(p, q, d, \varphi(n))$.

Théorème 2.1.1. *Si p, q sont premiers distincts, $n = pq$, $\gcd(e, \varphi(n)) = 1$, $d = e^{-1}[\varphi(n)]$, alors*

$$\begin{array}{rccc} f: & \mathbb{Z}/n\mathbb{Z} & \to & \mathbb{Z}/n\mathbb{Z} \\ & x & \mapsto & x^e[n] \end{array}$$

est bijective d'inverse

$$\begin{array}{rccc} f^{-1}: & \mathbb{Z}/n\mathbb{Z} & \to & \mathbb{Z}/n\mathbb{Z} \\ & x & \mapsto & x^d[n] \end{array}$$

2.1.2 Sécurité de RSA

Objectif de l'attaquant :

1. Trouver la clé secrète
2. Calculer $f^{-1}(y)$ pour certains y.

- Trouver $p, q, d, \varphi(n)$ à partir de n et e : déjà, si on connait p ou q, on peut facilement retrouver tout le reste de la clé. Ensuite si on connait $\varphi(n) = pq - (p+q) + 1$ et $pq = n$, donc $p + q = n - \varphi(n) + 1$, $pq = n$, et alors on peut trouver p, q. Enfin si on connaît d, alors $ed = 1[\varphi(n)]$. Prenons x aléatoire, on calcule $y = x^{\frac{ed-1}{1}}[n]$. Et alors $y^2 = x^{ed-1} = 1[n]$. Mais alors y est solution de l'équation $y^2 = 1[n]$, qui a 4 solutions : $(\pm 1, \pm 1) \in \mathbb{Z}/p\mathbb{Z} \times \mathbb{Z}/q\mathbb{Z}$ au travers du théorème des restes chinois. Mais alors si y correspond à $(1, -1)$ ou $(-1, 1)$ (notons $\alpha, -\alpha$ les éléments correspondants dans $\mathbb{Z}/n\mathbb{Z}$), alors $\gcd(y - 1, n) = p$ ou q (vu que $\alpha = 1[p]$ et $\alpha = -1[q]$).
- Trouver $f^{-1}(y)$ pour certains y (problème de la racine e-ième modulo n) : si on sait factoriser, on sait résoudre le problème de la racine e-ième grâce au théorème des restes chinois. On ne sait cependant pas si savoir résoudre le problème des racines e-ièmes nous permettrait de résoudre facilement le problème de factorisation.

2.2 RSA en signature

2.2.1 Problématique

diagramme 1

Algorithme de signature naif

On signe avec $f^{-1}(M) = M^d[n]$. Déjà, on doit supposer que $0 \leq M < n$ car sinon on aurait plusieurs messages avec la même signature.

- Si M est grand, on pourrait écrire $M = \sum M_k n^k$ avec $M_k \in [0, n-1]$, puis on signe par blocs, pas terrible ... diagramme 2
- Problème 2 : Si Alice envoie deux messages signés (M, S) et (M', S'), alors charlie peut signer MM' en calculant SS'.
- Problème 3 : Si Alice envoie un message M, S, alors charlie peut envoyer M^2, S^2, $\lambda^e M, \lambda S$.
- Prolbème 4 : charlie peut envoyer $(0, 0)$, $(1, 1)$, (λ^e, λ).

Paradigme "hash and sign"

diagramme 3

2.2.2 Fonctions de hachage

$h : \{0,1\}^* \to \{0,1\}^l$ avec $\{0,1\}^* = \sqcup_{n \geq 0} \{0,1\}^n$ et l un entier fixé.

Définition 2.2.1. Une telle fonction $h : \{0,1\}^* \to \{0,1\}^l$ est appelée fonction de hachage si elle vérifie les 3 propriétés suivantes :

p_1 **:** h est à sens unique, i.e. pour $y \in \{0,1\}^l$, il est calculatoirement difficile de trouver un antécédent x et y.

p_2 **:** h est à collisions faibles difficiles (second preimage resistant) i.e. pour $x \in \{0,1\}^*$ et $y = h(x)$, il est calulatoirement difficile de trouver $x' \in \{0,1\}$ tel que $x \neq x'$ et $h(x') = y$.

p_3 **:** h est à collisions fortes difficiles (collision resistant) i.e. il est calculatoirement difficile de trouver $x, x' \in \{0,1\}^*$ tel que $x \neq x'$ et $h(x) = h(x')$.

Rq 2.2.1. $p_2 \Rightarrow p_1$, $p_3 \Rightarrow p_2$. Ainsi d'un point de vu mathématique, p_3 suffit, mais il est intéressant de les écrire vu qu'elles ont un intérêt cryptographique.

1. Pour la propriété p_1, on a un algo qui trouve un antécédent par recherche exhaustive (on tire aléatoirement x et on regarder si $h(x) = y$). La complexité est en 2^l.
2. On peut faire la même chose pour la propriété p_2.
3. On génère aléatoirement $x_1, x_2, \cdots$, et on calcule leurs images $y_i = h(x_i)$ jusqu'à trouver une égalité du type $y_i = y_j$.