Cours : complexité

Christophe Ritzenthaler

9 septembre 2017

Rapport 2015 : "La ligne directrice du calcul formel est la recherche de l'effectivité, puis de l'efficacité (souvent en temps, parfois en espace) du calcul algébrique,..."

"Plus largement, une réflexion minimale sur les ordres de grandeur (est-ce qu'un calcul faisable représente 10^1, 10^{10}, 10^{100}, 10^{1000} opérations élémentaires) permettrait souvent de mieux situer les problèmes soulevés par un texte, ou de proposer des valeurs de paramètres réalistes quand ce sujet n'est pas évoqué par le texte."

"Les candidats ayant le réflexe de se saisir, seuls, d'une question de complexité, sont perçus très positivement par le jury."

"Estimation de la complexité des algorithmes précités dans le pire des cas. Aucune formalisation d'un modèle de calcul n'est exigée."

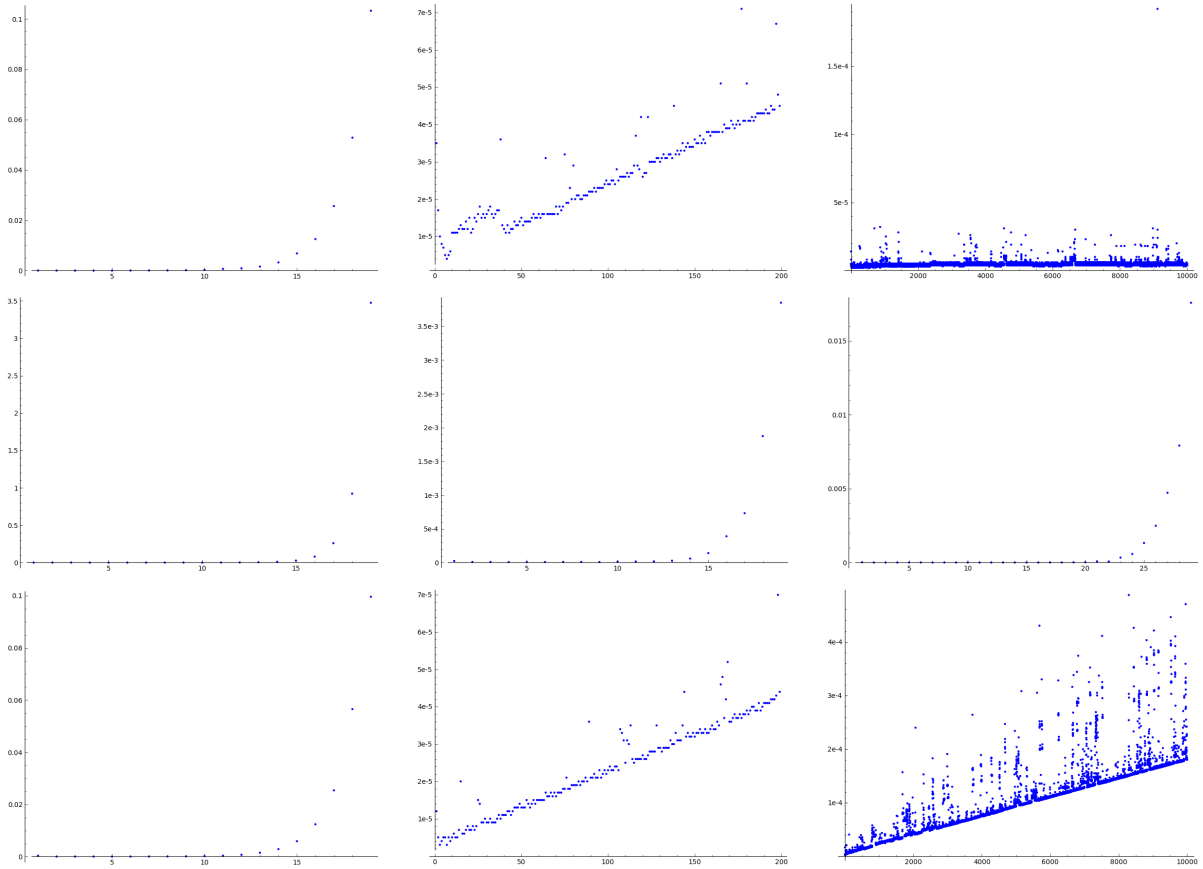
1 Quelques notions générales

1.1 Un exemple

Soit $(G, +)$ un groupe, $a \in G$ un élément fixe et n un entier. Calculer na. On veut mesurer le temps d'exécution de l'algorithme en fonction de la taille de l'entrée n. La taille d'un entier est le nombre de chiffres pour l'écrire. Le temps d'exécution dépend de la manière dont sont effectuées les opérations pour aboutir au résultat na. L'algorithme basique correspond à additionner $n-1$ fois a au résultat précédent en commençant par a et donc demande $n-1$ opérations dans G. C'est le premier niveau d'évaluation de la complexité. Pour obtenir une vraie évaluation du temps, il faut se demander combien coûte une opération dans G et cela dépend donc fortement du groupe en question et de la manière dont sont effectuées ces opérations. Distinguons plusieurs exemples

1. $G = (\mathbb{Z}, +)$ et
   - l'algorithme correspond à une boucle additionnant $n-1$ fois le résultat précédent.
   - l'algorithme correspond à faire $s_{i+1} = s_i + s_i$ dans le cas où n est une puissance de 2 (cas particulier de l'algorithme d'exponentiation rapide).
   - $n \cdot a$.
2. $G = (\mathbb{Q}^*, \times)$ et
   - l'algorithme correspond à une boucle multipliant $n-1$ fois le résultat précédent par a.
   - l'algorithme correspond à faire $s_{i+1} = s_i \cdot s_i$ dans le cas où n est une puissance de 2 (cas particulier de l'algorithme d'exponetiation rapide).
   - a^n
3. $G = (\mathbb{Z}/N\mathbb{Z}, +)$ et l'algorithme correspond à
   - une boucle additionnant $n-1$ fois le résultat précédent.
   - l'algorithme correspond à à faire $s_{i+1} = s_i \cdot s_i$ dans le cas où n est une puissance de 2 (cas particulier de l'algorithme d'exponetiation rapide).
   - a^n

On obtient les courbes suivantes.

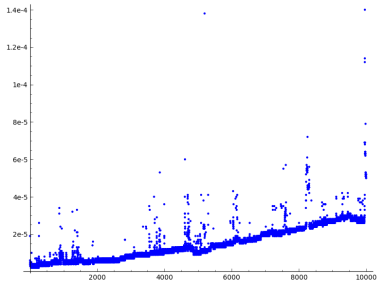


Dans les trois premiers cas, on a demandé de calculer $2^e \cdot 3$ avec e en abscisse :

— sur le dessin de gauche, on observe un comportement exponentiel qui provient de la boucle principale que l'on effectue 2^e fois.

— sur celui du milieu, on effectue e fois des opérations d'addition dont le coût est linéaire en la taille des éléments, c.-à-d. leur nombre de chiffres. Dans le cas présent le nombre de chiffres à chaque étape i est $\log_{10} 2^i \cdot 3 = \text{constante} \cdot i$ et donc le coût total devrait être $\sum_{i=1}^{n} \text{constante} \cdot i = O(n^2)$. Ce n'est pas ce qu'on observe. On peut donc penser que SAGE effectue les calculs en interne en base 2 et que donc le doublement à chaque étape est simplement un décalage qui est donc de coût constant.

— celui de droite provient probablement de la représentation en base 2 des opérations. En effet si on remplace 2^e par 3^e on obtient alors un coût linéaire qui s'explique par la manière dont est effectué le calcul de 3^e.



Pour les trois suivants, on a calculé 2^{2^e} avec e en abscisse et dans les dernier cas on a calculé $2^{2^e} \pmod{101}$ avec e en abscisse. On observe à chaque fois un comportement exponentiel. Ceci n'est pas surprenant car la sortie est de taille exponentielle. Simplement pour écrire la sortie, il faut donc un temps exponentiel en e.
Dans les trois derniers, on s'affranchit donc de cette question de taille et on obtient des résultats conforment à ce qu'on pourrait espérer. Notons toutefois que nous ne savons pas comment sont réalisés les opérations internes à $\mathbb{Z}/N\mathbb{Z}$ mais que $N = 101$ étant constant, elles sont d'un coût constant. La dépendance en N est une autre histoire.

Règles générales :

1. Le temps d'exécution d'une affectation ou d'un test est considéré comme constant ;
2. Le temps d'une séquence d'instruction est la somme des temps des instructions qui la composent ;
3. le temps d'un branchement conditionnel est égal au temps des maximum des alternatives
4. Le temps d'une boucle est égal au produit du temps des instructions contenues dans la boucle par le nombre de passages dans celle-ci.

Remarquons qu'on ne se ramène pas toujours à une complexité en nombres d'opérations élémentaires. Il est possible de s'arrêter à mi-chemin et de donner les résultats en fonctions par exemple du nombre d'additions, de multiplications,... Ceci permettra d'adapter le calcul final de la complexité en fonction des algorithmes pour ces opérations qui dépendent de la plage de tailles des données que l'on considèrera.

1.2 Principes

Le temps d'exécution d'un programme dépend de plusieurs données :

1. du type d'ordinateur utilisé ;
2. du langage utilisé (représentation des données) ;
3. de la complexité abstraite de l'algorithme sous-jacent.

Pour le mesurer en SAGE, on a vu qu'on peut utiliser les commandes suivantes

1. `t=cputime()` (on stocke dans t le temps CPU des opérations à venir)
2. succession d'opérations dont le temps est estimé
3. `cputime(t)` (on donne la valeur courante de t).

En général, on veut s'abstraire des deux premières données. Pour se faire, il faut être en mesure de donner un cadre théorique à l'algorithmique. Pour le premier point ceci se fait par la définition d'un ordinateur 'universel' (une *machine de Turing*) qui bien qu'extrêmement simple peut reproduire le comportement de n'importe quel ordinateur existant. Cela permet aussi de montrer que tout n'est pas "algorithmisable". Pour s'abstraire du second point, on regardera des classes d'équivalence de complexité (voir plus bas) plutôt que la complexité elle-même, ce qui permettra de ne pas se préoccuper des constantes qui interviennent dans les changements de représentations 'naturelles' et dans la définition des opérations élémentaires.

La mesure de la complexité d'un algorithme c'est :

1. évaluer les ressources (mémoire et CPU) utiles ;
2. Comparer deux algorithmes pour le même problème ;
3. donner une borne sur ce qui est effectivement possible de résoudre. On considère aujourd'hui qu'on peut réaliser en temps raisonnable 2^{60} opérations voire 2^{80} (extraction des bit-coins par an dans le monde). Quant à la mémoire elle est de l'ordre de 10^{10} octets.

On peut donc s'intéresser à deux types de complexité : en temps et en mémoire. Nous ne considérerons que la première. On considère aussi plusieurs mesures : dans le pire des cas, en moyenne ou dans le meilleur cas. Ici encore, on regardera surtout la complexité dans le pire des cas puisque c'est elle qui nous assure que l'algorithme se terminera toujours avec le temps au plus annoncé. Remarquons tout de même que les autres complexités peuvent aussi être utiles : par exemple, en cryptographie, c'est le temps minimal dans le meilleur des cas qui permet de décider si aucun algorithme ne pourra casser le protocole en un temps raisonnable.

Exemple 1. *Qu'est ce que la taille des entrées. Théoriquement, il s'agit du nombre de cases pour l'encodage de la donnée dans la machine de Turing. En pratique, on considèrera une mesure qui lui est "proportionnelle".*

- *Pour un entier, il s'agit du nombres de chiffres nécessaires à son écriture. On peut bien sûr imaginer des circonstances où d'autres facteurs seraient plus importants. Par exemple si un algorithme factorise un nombre aléatoire de plus en plus grand à chaque fois qu'il rencontre un* 9 *dans l'écriture du nombre. On notera* $|a|_b$ *la taille de* a *en base* b*. On a bien sûr* $|a|_b = \lfloor \log_b a \rfloor + 1$*. Remarquons qu'un changement de base multiplie la taille par une constante.*
- *Pour une liste, le paramètre est le cardinal de la liste. On peut noter que ceci est cohérent avec la complexité pour les entiers, car le tableau pourrait contenir les chiffres du nombre.*
- *Dans l'algorithme de la division euclidienne, on a deux entiers* a, b *en entrée. On mesurera la complexité en fonction du* $\sup(|a|, |b|)$*.*
- *Pour une matrice carrée de taille* n *c'est* n *le paramètre. Dans le cas d'opérations sur les matrices, la complexité sera alors exprimée comme une fonction de* n *des opérations concernant les coefficients (qui n'auront pas le même coût élémentaire*

selon la nature des coefficients -la complexité en fonction des opérations élémentaires pouvant être alors calculée dans un deuxième temps).
Lorsqu'on manipule des réels, la situation est également compliquée : il faudra définir une écriture tronquée pour pouvoir réaliser les calculs. Mais attention alors aux erreurs d'arrondis !
— *Pour un polynôme, c'est son degré.*
— *Enfin dans certains cas, il n'est pas possible de ne faire intervenir qu'une donnée. Par exemple pour un graphe, le nombre de sommets et d'arêtes peuvent être tous les deux des facteurs importants et pourtant complètement indépendants.*

Il est parfois très difficile d'évaluer les constantes en jeux. C'est pourquoi on distinguera plutôt des classes de complexité : si n est la taille de l'entrée, on dira que l'algorithme est polynomial (resp. exponentiel) si son temps d'exécution est de l'ordre de $\Theta(n^b)$ où $b \geq 1$ (resp. $\Theta(\exp(an))$ avec $a > 0$). Rappelons que deux fonctions f, g à valeurs positives sont du même ordre ($f = \Theta(g)$) si pour x assez grand, il existe deux constantes a, b telles que $af(x) \leq g(x) \leq bf(x)$. En pratique, on utilisera plutôt la comparaison $f = \mathcal{O}(g)$ pour simplement donner une majoration.

Attention toutefois aux limites d'un résultat asymptotique : pour une plage de données fixées (et finies) il se peut qu'un algorithme en $\mathcal{O}(Cn^2)$ soit plus rapide qu'un algorithme en $\mathcal{O}(C'n)$ si la constante C est beaucoup plus petite que la constante C'. C'est par exemple le cas de la multiplication par l'algorithme FFT (qui devient utile pour la multiplication dans $\mathbb{Z}$ à partir de 10000 chiffres binaires et pour les polynômes à partir du degré 10000).

Ces deux classes séparent les algorithmes dits efficaces (=polynomial) des autres (=exponentiel). En effet supposons que pour un nombre entier de taille n le programme A soit de complexité n^3 et le programme B de complexité 2^n. Alors si la puissance de calcul est multipliée par 1000, A pourra traiter des données $n' = 10n$ fois plus grande alors que le programme B seulement $n' \approx n + 10$.

2 Le cas du tri des listes

Un problème fondamental est celui du tri d'une liste de n d'entiers. Il est utile pour chercher plusieurs ($> \log n$) éléments, pour certains algorithmes (de chemins minimaux dans les graphes par exemple). On donnera la complexité en fonction du nombre de comparaisons à effectuer. Commençons par un résultat théorique.

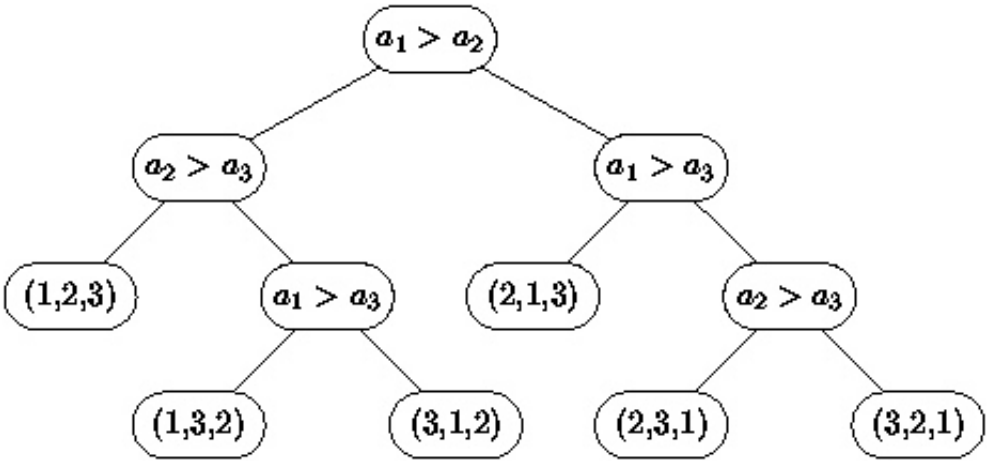
Théorème 2.1. *La complexité dans le pire des cas est minorée par* $\lceil \log_2 n! \rceil \sim \mathcal{O}(n \log n)$.

Démonstration. Présentons la preuve sous forme d'un exercice.

1. De combien de manières peut-on permuter les éléments d'un tableau $[a_1, a_2, \ldots, a_n]$?

2. Supposons que l'on applique un algorithme de tri donné à $[a_1, a_2, \ldots, a_n]$. Pour chaque comparaison effectuée au cours de l'algorithme, il y a deux possibilités : soit on change les éléments comparés, soit on ne les change pas.

   Supposons que, au maximum, l'algorithme s'exécute en $C(n)$ comparaisons ($C(n)$ est donc la complexité dans le pire cas). Montrer que $2^{C(n)} \geqslant n!$ en représentant l'algorithme par un arbre de décision. On s'inspirera de l'exemple ci-dessous pour le tri par insertion (voir ci-dessous) sur une liste à 3 éléments.



   En effet on peut modéliser un processus de tri par son arbre de décision. Chaque noeud est donné par la comparaison de deux éléments et suivant leur ordre, le noeud possède deux fils. Les feuilles contiennent la dernière comparaison réalisée par l'algorithme et un chemin descendant correspond à une exécution de l'algorithme. La profondeur de l'arbre est donc sa complexité dans le pire des cas en nombre de comparaisons. Un tel arbre contient au plus $2^{C(n)}$ feuilles. Mais puisque la liste peut avoir n'importe quel ordre, il faut que les $n!$ permutations apparaissent comme parmi les résultats de l'algorithme (une exécution d'un tri peut-être donnée par une unique permutation). Donc $n! \leq 2^{C(n)}$

3. Montrer qu'alors asymptotiquement, la complexité dans le pire cas est supérieure à un $O(n \log n)$. *Il faut se souvenir ici que* $\log n! = \sum_{i=1}^{n} \log i \leq n \log n$.

□

2.1 Tri par selection

```
1  def maxi(liste):
2      resultat = 0
3      for i in range(1, len(liste)):
4          if liste[i] > liste[resultat]:
5              resultat = i
6      return resultat
7
8  def tri_selection(A):
9      B=A
10     for i in range(0,len(A)-1):
```

```
11          t=maxi(B[0:len(A)-i])
12          b=B[len(A)-i-1]
13          B[len(A)-i-1]=B[t]
14          B[t]=b
15      return B
```

Le nombre de comparaisons est toujours $n(n-1)/2 = \mathcal{O}(n^2)$.

2.2 Tri-fusion

```
1  def fusion(A,B):
2      Ab=A
3      Bb=B
4      L=[]
5      while len(Ab)>0 and len(Bb)>0:
6          Ab[0]
7          if Ab[0]>Bb[0]:
8              L.append(Bb[0])
9              Bb.pop(0)
10         else:
11             L.append(Ab[0])
12             Ab.pop(0)
13     if len(Ab)==0:
14         return L+Bb
15     if len(Bb)==0:
16         return L+Ab
```

On effectue ici la fusion de deux listes triées en comparant à chaque fois le premier éléments de chaque liste. On effectue donc $\#A + \#B - 1$ comparaisons lorsque les listes sont de même taille. On effectue alors une procédure récursive.

```
1  def tri_fusion(A):
2      n=len(A)
3      if n<2:
4          return A
5      k=floor(n/2)
6      A1=A[:k]
7      A2=A[k:]
8      Ab=tri_fusion(A1)
9      Bb=tri_fusion(A2)
10     return fusion(Ab,Bb)
```

On peut montrer le résultat suivant

Proposition 2.1. *La complexité exacte du tri fusion est* $n\lceil \log_2 n \rceil - 2^{\lceil \log_2 n \rceil} + 1$.

Démonstration. Soit $C(n)$ la complexité (dans le cas moyen ou le pire, c'est la même). On a alors

$$C(n) = n - 1 + C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil).$$

On montre que la fonction introduite dans l'énoncé satisfait la récurrence (on pourrait montrer qu'il y a unicité si on fixe les valeurs en 1 et 2). On distingue le cas pair et le cas impair. Le cas difficile est le cas impair $n = 2k + 1$. On a

$$C(2k+1) = 2k + C(k) + C(k+1).$$

On distingue le cas général où $2^r < k < 2^{r+1}$ et le cas $k = 2^r$. Dans le premier cas

$$\begin{aligned} A_1 &= (2k+1)\lceil \log_2(2k+1)\rceil - 2^{\lceil \log_2(2k+1)\rceil} + 1 \\ &= (2k+1)(\lceil \log_2 k\rceil + 1) - 2\cdot 2^{\lceil \log_2(k)\rceil} + 1 \\ &= 2k+2+(2k+1)\lceil \log_2 k\rceil - 2\cdot 2^{\lceil \log_2(k)\rceil} \end{aligned}$$

$$A_2 = k\lceil \log_2 k\rceil - 2^{\lceil \log_2 k\rceil} + 1$$

$$A_3 = (k+1)\lceil \log_2(k)\rceil - 2^{\lceil \log_2(k)\rceil} + 1$$

car $\lceil \log_2(2k+1)\rceil = \lceil \log_2(k)\rceil + 1 = \lceil \log_2(k+1)\rceil + 1 = r+2$. On a bien l'égalité $2k + A_2 + A_3 = A_1$.
Dans le second cas, on a $\lceil \log_2(2k+1)\rceil = r+2$, $\lceil \log_2(k)\rceil = r$ et $\lceil \log_2(k+1)\rceil = r+1$. On a alors

$$A_1 = (2^{r+1}+1)(r+2) - 2^{r+2} + 1 = 2^{r+1}r + r + 3,$$

$$A_2 = 2^r r - 2^r + 1$$

et

$$A_3 = (2^r+1)\cdot(r+1) - 2^{r+1} + 1 = 2^r r - 2^r + r + 2.$$

On vérifie alors que $2^{r+1} + A_2 + A_3 = A_1$. □

Il s'agit donc d'un algorithme asymptotique optimal.

Remarque 1. *Le tri-fusion permettent d'atteindre la complexité* $\lceil \log_2 n!\rceil$ *lorsque* $n = 1, 2, 3, 4$. *Par contre pour* $n = 5$, *on a une complexité de* 8 *comparaisons alors qu'il existe bel et bien un algorithme en* 7 *comparaisons. Pour* $n = 12$, *la borne minimale est de 29 mais on a montré qu'on ne pouvait pas faire mieux que 30. La valeur minimale pour tout* n *est inconnue à l'heure actuelle.*

En pratique, il est toutefois utile d'avoir une manière plus simple d'estimer les complexités des programmes récursifs. Ceci est contenu dans le théorème du Maître (ici une version très faible).

Proposition 2.2. *Soit* C *une fonction de coût croissante satisfaisant à* $C(n) \leq mC(n/p) + T(n)$ *pour tout* $n = p^e$ *puissance positive de* p *et* $C(1) = \kappa$. *On suppose de plus que* $T(n) \leq a \cdot n^k$ *et que c'est une fonction positive. On a alors*

1. $C(n) = O(n^k)$ *si* $m < p^k$;
2. $C(n) = O(n^k \log n)$ *si* $m = p^k$;
3. $C(n) = O(n^{\log_p m})$ *si* $m > p^k$;

Démonstration. Par récurrence, on a

$$C(n) \leq T(n) + mT(n/p) + m^2 T(n/p^2) + \ldots m^{e-1}T(p) + m^e\kappa.$$

Hence

$$C(n) \leq an^k \sum_{i=0}^{e-1} (m/p^k)^i + m^e \kappa.$$

On distingue alors selon les cas

1. Si $m < p^k$ alors la somme est bornée et on obtient donc une complexité en $O(n^k)$ (car $m^e \leq p^{ke} = n^k$).
2. Si $m = p^k$ alors la somme est constante et vaut $e = \log_p n$, d'où le résultat.
3. Si $m > p^k$ alors on réécrit

$$C(n) \leq an^k \cdot (m/p^k)^{e-1} \sum_{i=0}^{e-1} (p^k/m)^i + m^e \kappa.$$

On a donc de nouveau une somme bornée. Le terme principal est

$$n^k \cdot (m/p^k)^{e-1} = n^k (m/p^k)^e (p^k/m) = m^{e-1} p^k \leq m^e = exp(\log m \log n / \log p) = n^{\log_p m}.$$

□

3 Algorithmes en calcul formel

3.1 Évaluation d'un polynôme

Soit $P(X) = a_n X^n + \ldots + a_1 X + a_0 \in \mathbb{R}[X]$. Nous allons nous préoccuper du nombre d'opérations élémentaires (multiplication et addition) nécessaires pour évaluer P en un nombre réel. Si on effectue le calcul de manière basique, on a $2n - 1$ multiplications et n additions.

3.1.1 Méthode de Horner

Ecrivons

$$a_n x^n + \ldots + a_1 x + a_0 = ((\ldots (a_n x + a_{n-1})x + a_{n-2}) \ldots) x + a_0.$$

On obtient n multiplications et additions.

3.1.2 Méthode du produit

Si $P(x) = a_0 \prod (x - \alpha_i)$ avec tous les α_i réels, on a également besoin de n multiplications et additions.

3.1.3 Forme orthogonale

On écrit $P(x) = \sum_{k=0}^{n} b_k Q_k(x)$ où les polynômes Q_i satisfont

$$Q_{i+1} = (A_i x + B_i)Q_i(x) - C_i Q_{i-1}(x), \quad A_i \neq 0,\ Q_0 = 1,\ Q_{-1} = 0.$$

L'évaluation requiert $3n - 1$ multiplications et additions. Parmi les familles de polynômes orthogonaux, on a les polynômes de Chebyshev, T_n qui sont le meilleur choix en terme d'efficacité de l'évaluation. Rappelons que les polynômes de Chebyshev satisfont $T_n(\cos(x)) = \cos(nx)$ et sont donnés par la récurrence $A_i = 2$ pour $i \geq 1$, $A_0 = 1$ et $B_i = 0, C_i = 1$ pour $i \geq 0$.

3.1.4 Pré-processing

Si on désire évaluer le polynôme en plusieurs points, il peut être intéressant de s'autoriser une modification des coefficients avant le début des calculs afin de simplifier ces derniers. Ceci s'appelle le *pré-processing*. Considérons le cas d'un polynôme de degré 4. On écrit

$$\begin{aligned} a_4x^4 &+ a_3x^3 + a_2x^2 + a_1x + a_0 \\ &= a_4((x(x+\alpha_0)+\alpha_1)(x(x+\alpha_0)+x+\alpha_2)+\alpha_3) \qquad (1) \\ &= a_4x^4 + a_4(2\alpha_0+1)x^3 + a_4(\alpha_1+\alpha_2+\alpha_0(\alpha_0+1))x^2 \\ &+ a_4((\alpha_1+\alpha_2)\alpha_0+\alpha_1)x + a_4(\alpha_1\alpha_2+\alpha_3). \end{aligned}$$

On peut alors trouver les α_i en fonction des coefficients a_i. La forme de (**??**) utilise 3 multiplications et 5 additions. Si, comme c'est souvent le cas, les multiplications sont plus coûteuses que les additions, cet algorithme présente donc un avantage sur la méthode de Horner.

3.1.5 Quelques résultats théoriques

De nombreux résultats sont connus sur ce qu'on peut faire de mieux pour ce problème.

Théorème 3.1. *Tout algorithme évaluant un polynôme P de degré n sans pré-processing a besoin d'au moins n multiplications et n additions.*

Ainsi dans ce cas l'algorithme de Horner est optimal.

Au contraire si on accepte le pré-processing, on a le résultat suivant.

Théorème 3.2. *Il existe un algorithme explicite (de Belaga) pour l'évaluation d'un polynôme en $\lfloor (n+1)/2 \rfloor + 1$ multiplications et $n+1$ additions. Il n'existe pas d'algorithme avec moins de $\lfloor (n+1)/2 \rfloor$ multiplications ou moins de n additions.*

On ne sait pas si le seuil peut être atteint ou non. L'algorithme de Belaga peut avoir besoin de coefficients complexes. Il en existe un autre (l'algorithme de Pan) en $\lfloor n/2 \rfloor + 2$ multiplications et $n+1$ additions qui pour des coefficients de P 'naturels' a des paramètres réels. Par la suite Rabin et Winograd ont donné une méthode en $\lceil n/2 \rceil + \mathcal{O}(\log n)$ multiplications/divisions et $n + \mathcal{O}(n)$ additions avec uniquement des coefficients réels.

3.1.6 Une question auxiliaire : la stabilité et le conditionnement

Grossièrement, un algorithme est dit *numériquement instable* s'il crée des erreurs d'arrondis qui augmentent de manière incontrôlé. Un autre concept est le *mauvais conditionnement*. Un algorithme est mal conditionné si une petite incertitude sur les données d'entrées peut créer une grande incertitude sur le résultat. Le tableau ci-dessous résume ces propriétés pour les algorithmes d'évaluation.

Forme	Conditionnement	stabilité
Horner	peut être mal conditionné	peut être instable
Forme produit	assez bien conditioné	stable
Chebyshev	bien conditionné	stable
Belaga	peut être mal conditionné	peut être très instable
Pan	Conj. : si petits Pan coef. alors bien conditionné	peut être très instable.

3.2 Exponentiation rapide

Soit $(G, \times)$ un groupe, $x \in G$ et $n \in \mathbb{N}$. On souhaite calculer x^n en minimisant le nombre de multiplications. La méthode naïve demande $n-1$ multiplications et c'est donc une méthode exponentielle en la taille de n. On peut faire beaucoup mieux avec *l'exponentiation rapide*.

3.2.1 Méthodes binaires

Écrivons $n = (n_{\ell-1}, \dots, n_0)$ en base 2. Cette méthode est basée sur l'égalité

$$x^{(n_{\ell-1},\dots,n_i)_2} = (x^{(n_{\ell-1},\dots,n_{i+1})_2})^2 \cdot x^{n_i}.$$

Cette méthode est dite de *gauche vers la droite* (par ordre de traitement des n_i et se traduit par l'algorithme suivant.

```
1  def leftoright(x,n):
2      m=n.binary()
3      l=len(m)
4      y=1
5      i=0
6      while i <l:
7          y=y^2
8          if m[i]=='1':
9              y=x*y
10         i=i+1
11     return y
```

Ce premier algorithme a l'avantage de faire une multiplication fixe par x qui peut donc être optimisée.
Une autre méthode opère de droite à gauche et repose sur

$$x^{(n_i,\dots,n_0)_2} = x^{(n_{i-1},\dots,n_0)_2} \cdot x^{n_i 2^i}.$$

```
1  def rightoleft(x,n):
2      m=n.binary()
3      l=len(m)
```

```
    y=1
    z=x
    i=l-1
    while i >-1:
        if m[i]=='1':
            y=y*z
        z=z^2
        i=i-1
    return y
```

Ce second algorithme a l'avantage de pouvoir calculer l'écriture en base 2 de m simultanément puisque on regarde à chaque fois la parité. On obtient par exemple

```
def rightoleft(x,n):
    y=1
    z=x
    m=n
    while m >0:
        if m%2==1:
            y=y*z
            m=(m-1)
        m=m/2
        z=z^2
    return y
```

Exemple 2. *Donnons un exemple x^{314}. On a $314 = (100111010)_2$ et $\ell = 9$. Premier algorithme*

i	8	7	6	5	4	3	2	1	0
n_i	1	0	0	1	1	1	0	1	0
y	x	x^2	x^4	x^9	x^{19}	x^{39}	x^{78}	x^{157}	x^{314}

Second algorithme

i	0	1	2	3	4	5	6	7	8
n_i	0	1	0	1	1	1	0	0	1
z	x	x^2	x^4	x^8	x^{16}	x^{32}	x^{64}	x^{128}	x^{256}
y	1	x^2	x^2	x^{10}	x^{26}	x^{58}	x^{58}	x^{58}	x^{314}

Dans les deux cas, il faut faire $l - 1 \simeq \log_2 n$ carrés et au pire le même nombre de multiplications. On obtient donc un algorithme en $O(\log n)$ multiplications dans G. De nombreuses variantes existent qui permettent de réduire le nombre de multiplications mais ne diminuent pas la complexité asymptotique.

3.3 Division euclidienne de polynômes

3.3.1 Calcul rapide sur les séries formelles

Soit A un anneau commutatif unitaire, on note $R = A[[X]]$ l'anneau des séries formelles sur A. Ses éléments sont des suites $(f_i)_{i\in\mathbb{N}}$ d'éléments de A notées

$$F(X) = \sum_{i\geq 0} f_i X^i$$

et munies des deux opérations

$$\sum_{i\geq 0} f_i X^i + \sum_{i\geq 0} g_i X^i = \sum_{i\geq 0} (f_i + g_i) X^i$$

et

$$\sum_{i\geq 0} f_i X^i \cdot \sum_{i\geq 0} g_i X^i = \sum_{i\geq 0} (\sum_{a+b=i} f_a g_b) X^i.$$

En utilisant le fait que si $F = 1 + GX$ alors $H = 1 - GX + (GX)^2 - \dots$ est tel que $FG = 1$, il est facile de voir que les éléments inversibles de R sont les séries formelles de terme constant inversible dans A. On se propose de donner ici un algorithme inspiré de la méthode de Newton pour le calcul des racines afin de calculer rapidement un "approximation" de l'inverse d'un série formelle inversible.
Étant donné deux séries formelles F, G on dit que $F - G = O(X^n)$ si les termes de F et de G coïncident au moins jusqu'à l'ordre $n-1$. On appellera alors approximation de F (à l'ordre n, la série tronquée donnée par le polynôme $G = \sum_{i=0}^{n-1} f_i X^i$. Bien sûr, en machine, on ne peut manipuler que des séries tronquées.

Remarque 2. *On peut aussi munir R d'une distance. Notons $v(F) = \{\inf i,\ f_i \neq 0\}$. Alors $d(F,G) = 2^{-v(F-G)}$ est une distance. On voit que la troncature permet de définir également une approximation au sens de la distance ci-dessus.*

Lemme 3.1. *Soit $F \in R$ une série formelle de terme constant inversible et G une série telle que $G - 1/F = O(X^n)$ avec $n > 0$. Alors*

$$N(G) = G + G(1 - GF)$$

vérifie $N(G) - 1/F = O(X^{2n})$.

L'itération N provient de la méthode de Newton $y_{i+1} = y_i - \frac{\Phi(y_i)}{\Phi'(y_i)}$ appliquée à la fonction $\Phi(u) = 1/u - F$.

Démonstration. Par hypothèse on peut définir $H \in R$ tel que $GF = 1 - HX^n$. On a alors

$$1/F = G(1-HX^n)^{-1} = G(1+HX^n+O(X^{2n})) = G(1+HX^n)+GO(X^{2n}) = N(G)+O(X^{2n}).$$

□

Ceci induit donc l'algorithme récursif suivant

```
1 R.<T> = PowerSeriesRing(QQ)
2 def inverse(f,N):
3     if N==1:
4         return f[0]^(-1)
5     F=f.truncate(N)
6     G=inverse(F,ceil(N/2))
7     G2=G+(1-G*F)*G
8     return G2.truncate(N)
```

La complexité $C(N)$ de l'algorithme peut être facilement exprimer avec le théorème du Maître. En effet,

$$C(N) \leq C(N/2) + 2M(N) + aN$$

où $M(N)$ est le coût (arithmétique) de la multiplication de deux polynômes de degré au plus N et a une constante pour le coût des additions. Rappelons qu'il vaut N^2 pour la méthode naïve $N^{\log_2 3}$ par Karatsuba et $N \log N \log\log N$ par la FFT. Dans tous les cas on obtient $C(N) = O(M(N))$.

3.3.2 Division euclidienne

La méthode naïve de division euclidienne d'un polynôme $F \in A[X]$ par un polynôme $G \in A[X]$ unitaire de degré inférieur consiste à écrire $F = aX^m + T_F$ et $G = X^n + T_G$ avec $\deg T_F \leq m$ et $\deg T_G \leq n$ et à faire $F - aX^{m-n}G$ qui est alors un polynôme de degré $< m$ et on réitère. La complexité est facile à estimer : la boucle élémentaire a au plus $O(n)$ opérations et il faut passer $m - n + 1$ fois au pire dans cette boucle, soit une complexité en $O(n(m-n))$. C'est un (bon) algorithme quadratique. Mais on peut obtenir un algorithme quasi-optimal de la manière suivante. L'égalité $F = QG + R$ peut se réécrire $F/G = Q + R/G$ et on peut donc voir Q comme le développement asymptotique de F/G à l'infini. On se ramène en 0 par le changement de variable $1/T$

$$\frac{T^m F(1/T)}{T^n G(1/T)} = T^{m-n} Q(1/T) + \frac{T^m R(1/T)}{T^n G(1/T)}.$$

Concrètement

1. Calculer $T^m F(1/T)$ et $T^n G(1/T)$ (en inversant l'ordre des coefficients)
2. Calculer le quotient $T^m F(1/T)/(T^n G(1/T)) \pmod{T^{m-n+1}}$ par une inversion de série formelle et un produit.
3. On en déduit Q car la valuation de $\frac{T^m R(1/T)}{T^n G(1/T)}$ est plus grande que $m - n$.
4. En en déduit R par $F - QG \pmod{X^n}$.

La complexité de l'algorithme est $O(M(n))$. Voici le code associé

```
def division(F,G):
    if F.degree()<G.degree():
        return [0,F]
    m=F.degree()
    n=G.degree()
    FR=T^m*F(1/T)
    GR=T^n*G(1/T)
    QR=inverse(GR,m-n+1)*(FR.truncate(m-n+1))
    Q=T^(m-n)*QR(1/T)
    R=F-Q*G
    return [Q,R]
```

3.3.3 Un exercice

Soit $\mathbb{A}$ un anneau commutatif. On note $\mathbb{A}(X)$ l'anneau des fractions rationnelles sur $\mathbb{A}$. Ses éléments sont de la forme A/B avec $A, B \in \mathbb{A}[X]$ et $B \neq 0$. On définit le degré

de cette fraction comme $\max(\deg(A), \deg(B))$. On souhaite calculer le développement en série formelle F lorsque $B(0)$ est inversible.

1. Montrer que deux fractions rationnelles de degré au plus n peuvent être additionnées et multipliées en temps $O(M(n))$.
2. Montrer qu'une fraction rationnelle est uniquement déterminée par les $\deg(A) + \deg(B) + 1$ premiers termes de son développement.
   Supposons qu'il existe $A'/B' \neq A/B$ telle que $A'/B' \equiv F \pmod{X^{m+n+1}}$ où $m = \deg(A) \geq \deg(A'), n = \deg(B) \geq \deg(B')$. On a alors $A'/B' - A/B \equiv 0 \pmod{X^{m+n+1}}$. Comme B et B' sont inversibles, cela signifique $A'B - B'A \equiv 0 \pmod{X^{m+n+1}}$. Comme $\deg(A'B - B'A) \leq m+n$, ceci impose $A'B = B'A$ d'où le résultat.
3. Montrer qu'on peut calculer le développement à l'ordre N d'une fraction rationnelle de degré $d \leq N$ en $O(M(N))$ opérations.
   On inverse B comme une série formelle puis on multiplie par A. On reste donc en $O(M(N))$ puisque $\deg(A) \leq N$.
4. Montrer qu'on peut se ramener au cas où $d = \deg(B)$ en $O(M(d))$ opérations.
   On commence par une division euclidienne de A par B. Ceci coûte $O(M(d))$ opérations.

On souhaite aboutir à une meilleure complexité et montrer le résultat suivant

Théorème 3.3. *Soit A/B de degré au plus d avec $B(0)$ inversible. Le développement en série formelle de A/B à précision $N \geq d$ peut se calculer en $O(NM(d)/d)$ opérations.*

Pour montrer le théorème on aura besoin du lemme suivant.

Lemme 3.2. *Soit A/B avec $B(0)$ inversible. Soit d le degré de B, $\deg(A) < d$ et $\kappa \geq 0$. Les coefficients $c_\kappa, \ldots, c_{\kappa+d-1}$ du développement de A/B sont les coefficients de $X^{2d-2}, \ldots, X^d, X^{d-1}$ du produit*

$$(X^\kappa \pmod{B(1/X)X^d})(c_{2d-2} + \ldots + c_0 X^{2d-2}).$$

On commence par montrer le lemme. Soit C la matrice de la multiplication par X dans la base $1, X, \ldots, X^{d-1}$ de $\mathbb{B} = \mathbb{A}[X]/(\bar{B})$ où $\bar{B} = B(1/X)X^d$.

5. Montrer qu'on a $(c_{n+1}, \ldots, c_{n+d}) = (c_n, \ldots, c_{n+d-1}) \cdot C$ pour tout $n \geq 0$ (on exploitera l'égalité $B \cdot F = A$ et le fait que $\deg(A) < d$).
   le dernier élément du produit $(c_n, \ldots, c_{n+d-1})C$ est $\frac{-1}{b_0} \sum_{i=}^{d-1} c_{n+i} b_{d-i}$. D'autre part $\sum b_i X^i \sum c_i X^i = A$ donc le coefficient de degré $n \geq d$ est nul. Or celui ci vaut $\sum_{i=0}^{d} c_{n+d-i} b_i = c_{n+d} b_0 + \sum_{i=0}^{d-1} c_n b_{d-i}$. On a donc le résultat.
6. En déduire un algorithme en $O(dN + M(d))$ pour calculer le développement à l'ordre N.
   le dernier élément du produit $(c_n, \ldots, c_{n+d-1})C$ est $\frac{-1}{b_0} \sum_{i=}^{d-1} c_{n+i} b_{d-i}$. D'autre part $\sum b_i X^i \sum c_i X^i = A$ donc le coefficient de degré $n \geq d$ est nul. Or celui ci vaut $\sum_{i=0}^{d} c_{n+d-i} b_i = c_{n+d} b_0 + \sum_{i=0}^{d-1} c_n b_{d-i}$. On a donc le résultat.

7. En déduire que $c_{\kappa+n}$ peut être obtenu à partir de $(c_n, \ldots, c_{d+n-1})$ et de C^κ.
On calcule les d premiers c_i par un algorithme de son choix. Les autres sont alors obtenus par la formule de récurrence qui demande $O(d)$ opérations à chaque étape. En particulier pour $d = 1$, on a un algorithme linéaire en $O(N)$.
8. *On a $(c_n, \ldots, c_{n+d-1}C^\kappa = (c_{n+\kappa}, \ldots, c_{n+d+\kappa-1})$. Donc on obtient $c_{n+\kappa}$ comme la première colonne de C^κ fois les coefficients de $(c_n, \ldots, c_{n+d-1})$.*
9. En utilisant l'interprétation de C comme multiplication par X, montrer que la première colonne de C^κ contient les coefficients du polynôme $X^\kappa \pmod{\bar{B}}$.
En effet C^κ est la multiplication par $X^\kappa \pmod{\bar{B}}$ donc la première colonne contient les coefficients dans la base $1, \ldots, X^{d-1}$ de l'image de 1, i.e. X^κ.
10. En déduire le lemme.
Si on note $\alpha_0, \ldots, \alpha_{d-1}$ les entrées de la première colonne de C^κ, on a en multipliant C^κ par le vecteur $(c_j, \ldots, c_{d-1+j})$ que $c_{\kappa+j} = \sum c_{i+j}\alpha_i$. D'autre part on a que $X^\kappa \pmod{\bar{B}} = \sum \alpha_i X^i$ et donc la multiplication par $c_{2d-2} + \ldots + c_0 X^{2d-2}$ donne pour le coefficient en X^{2d-2} : $\sum \alpha_i c_i$ et plus généralement pour le coefficient en X^{2d-2-j} : $\sum_{i=0}^{d-1} \alpha_i c_{i+j}$, d'où l'égalité.
11. Montrer qu'en utilisant $\lceil N/d \rceil - 2$ fois le lemme avec $\kappa = 2d, \ldots$, on peut calculer $c_{2d}, \ldots, c_N$.
Avec $\kappa = 2d$, on obtient en utilisant le lemme $c_{2d}, \ldots, c_{3d-1}$ puis avec $\kappa = 3d$, on obtient $c_{3d}, \ldots, c_{4d-1}$. Donc avec $\kappa = \lceil N/d \rceil d$, on obtient au moins $c_N, \ldots, c_{N+d-1}$.
12. *On calcule d'abord $y = X^d$ dans $\mathbb{B}$: ceci a un coût unité dans $\mathbb{B}$ car il suffit d'enlever le terme de tête car X^d est de degré d. Ensuite il faut faire $\lceil N/d \rceil$ multiplications par y. D'où le résultat.*
13. Montrer qu'on peut calculer les restes $X^{di} \pmod{\bar{B}}$ pour $0 \leq i \leq \lceil N/d \rceil$ en $O(N/d)$ opérations dans $\mathbb{B}$.
C'est le résultat sur le développement par Newton déjà utilisé à la question 3.
14. Montrer qu'on peut calculer $c_0, \ldots, c_{2d-2}$ en $O(M(d))$ opérations dans $\mathbb{A}$.
Chaque opération dans $\mathbb{B}$ coûte $O(M(d))$ donc on a $O(\lceil N/d \rceil M(d))$ opération dans $\mathring{A}$ pour le calcul des X^{di}. On a ensuite besoin de $O(M(d))\lceil N/d \rceil$ opération pour le produit du lemme et encore $O(d)$ opérations pour les premiers termes. D'où le résultat.
15. Conclure.

3.4 Racines carrées dans $\mathbb{F}_p$

3.4.1 Les algorithmes probabilistes

Il est utile de considérer une classe plus large d'algorithmes, dit *probabilistes*, car autorisant le recours à des jets aléatoires. Notons qu'en pratique l'aléatoire est difficile à obtenir. Il est soit obtenu par des moyens physiques (mesure de déplacements de la souris, variations de la température du processeur,...), ce qui est difficile et coûteux, ou en utilisant des algorithmes qui sont dit pseudo-aléatoires mais qui reste relativement lent (ou/et peu sûrs).

Exemple 3. — *La méthode de Von Neumann : prendre un nombre (ex.* 1111*), l'élever au carré (*$1111^2 = 1234321$*), récupérer les chiffres du milieu (*3532*), recommencer.*
— *générateur congruentiel linéaire :* $X_{n+1} \equiv aX_n + c \pmod{m}$*. Avec par exemple* $a = 16807$*,* $c = 0$ *et* $m = 2^{31} - 1$*.*
— $X_{n+1} \equiv X_n(X_n + 1) \pmod{2^e}$*,* $X_0 \equiv 2 \pmod{4}$ *;*
— $X_n \equiv X_{n-24} + X_{n-55} \pmod{m}$*.*
— *Blum Blum Shub.*

On renvoie à la page wikipédia sur les générateurs pseudo-aléatoires pour plus d'informations.

La classe des algorithmes probabilistes suppose qu'on peut obtenir de bons jets aléatoires pour un coût fixe et de ce fait contiennent des algorithmes plus efficaces que ceux déterministes. On distingue trois types :
— *Las Vegas* : peut ne pas s'arrêter mais s'il s'arrête donne la bonne réponse ;
— *Monte Carlo* : s'arrête toujours et si la réponse est affirmative alors elle est bonne mais peut se tromper dans le cas d'une réponse négative ;
— *Atlantic city* : s'arrête toujours et peut se tromper toujours mais la probabilité d'une mauvaise réponse est plus faible que la probabilité d'une bonne réponse.

Dans les deux derniers cas, ces algorithmes sont utilisés de manière itérative afin de faire diminuer la probabilité de l'erreur jusqu'à un seuil donné.

3.4.2 Un exemple d'algorithme de Las Vegas

Soit a un élément de $\mathbb{Z}/p\mathbb{Z}$, $p > 2$ premier qui est un carré. On veut calculer x tel que $x^2 \equiv a \pmod{p}$.
Quand $p = 4k - 1$ on vérifie que $a^{\frac{p+1}{4}}$ est une racine carrée de a

$$(a^{\frac{p+1}{4}})^2 \equiv a^{(p-1)/2} \cdot a \equiv x^{p-1} \cdot a \equiv a \pmod{p}.$$

Quand $p = 4k + 1$, on ne connaît pas d'algorithme rapide non probabiliste. On utilise alors l'algorithme de Shanks. On écrit $p - 1 = 2^s t$ avec t impair et $s \geq 2$. Soit $a \equiv x^2 \pmod{p}$. On suppose que l'on connaît b qui n'est pas un résidu quadratique modulo p. On peut le déterminer grâce à l'algorithme de Las Vegas suivant

```
1 def nonsquare(p):
2     a=1
3     while a.is_square()==true:
4         a=GF(p).random_element()
5     return a
```

On pose $z = b^t$. Soit $B = a^t$, $X = a^{(t+1)/2}$, $Y = z$ et $R = s - 1$.

```
1 def racine(a,p):
2     if p%4 ==3:
3         return a^((p+1)/4)
4     else:
5         s=(p-1).valuation(2)
6         t=(p-1)/2^s
```

```
7           B=a^t
8           X=a^((t+1)/2)
9           b=nonsquare(p)
10          z=b^t
11          Y=z
12          R=s-1
13          while R>0:
14              if B^(2^(R-1))%p ==1:
15                  Y=Y^2
16              else:
17                  B=B*Y^2
18                  X=X*Y
19                  Y=Y^2
20              R=R-1
21          return X
```

Pour vérifier que X est une racine carrée de a, on vérifie que les conditions suivantes sont des invariants de boucles :

$$\begin{cases} aB & = X^2 \\ Y^{2^R} & \equiv -1 \pmod{p} \\ B^{2^R} & \equiv 1 \pmod{p} \\ R & \geq 0 \end{cases}$$

Pour les conditions initiales

$$\begin{cases} aB & = aa^t = a^{t+1} = (a^{(t+1)/2})^2 \\ Y^{2^R} & \equiv z^{2^{s-1}} \equiv b^{(p-1)/2} \equiv -1 \pmod{p} \\ B^{2^R} & \equiv a^{t2^{s-1}} \equiv a^{(p-1)/2} \equiv 1 \pmod{p} \\ R = s-1 & \geq 0 \end{cases}$$

Après une étape, notons B', X', Y', R' les nouvelles valeurs.

— Si $B^{2^{R-1}} \equiv 1 \pmod{p}$, alors

$$\begin{cases} aB' & = aB = a^{t+1} = (a^{(t+1)/2})^2 \\ (Y')^{2^{R'}} & = Y^{2^R} \equiv -1 \pmod{p} \\ (B')^{2^{R'}} & \equiv B^{2^{R-1}} \equiv 1 \pmod{p} \\ R' = R-1 & \geq 0 \end{cases}$$

— Si $B^{2^{R-1}} \equiv -1 \pmod{p}$, alors

$$\begin{cases} aB' & = aBY^2 = X^2Y^2 = X'^2 \\ (Y')^{2^{R'}} & = Y^{2^R} \equiv -1 \pmod{p} \\ (B')^{2^{R'}} & \equiv B^{2^{R-1}}Y^{2^R} \equiv 1 \pmod{p} \\ R' = R-1 & \geq 0 \end{cases}$$

Le caractère probabiliste de l'algorithme vient du fait que pour obtenir rapidement b, on procède à des tirages aléatoires par les entiers de 2 à $p-1$. Remarquons que sous certaines conjectures, on peut rendre cet algorithme non probabiliste.