M1 MINT – Université Paris-Saclay

– MAO – Algèbre avec SAGE

N. Perrin

Université de Versailles Saint-Quentin-en-Yvelines
Année 2017-2018

Ce cours est une introduction au calcul formel avec une partie pratique qui utilise le logiciel SAGE.

Le calcul formel calcule des objets mathématiquement *exacts* par opposition au calcul aproché dont il est question dans le seconde partie de ce cours (second partie enseignée par Tahar Boulmezaoud). On va y considérer deux aspects importants du calcul formel

1. la “calculabilité” et
2. la “complexité”.

La calculabilité comme son nom l'indique cherche à déterminer si un calcul est réalisable par une machine. Il s'agira donc dans un premier temps de fournir un algorithme de calcul puis de l'implémenter, dans un laguage, afin de la faire tourner sur machine (et de vérifier qu'il fait bien ce qu'on lui demande !).

La complexité étudie les algorithmes obtenus à l'étape précédente et évalue le temps de calcul. On cherche bien sûr à réaliser des algorithmes qui soient les plus rapides possibles.

Table des matières

Première partie .

Arithmétique des grands entiers et des polynômes

1. Représentation des grands entiers

Avant même de pouvoir commencer à concevoir des algorithmes et écrire des programmes, un point imporant est de comprendre les objets que l'on manipule. Les plus simples sont les entiers. Mais attention, on a dit que l'on voulait faire des calculs exacts et ce quels que soient les entiers et notamment *quelque soit leur taille*. Ceci peut poser un problème comme le montre l'exemple suivant (cf. feuille de TD 1).

Exemple 1.0.1 On tape la commande suivante dans SAGE qui nous renvoie :

```
sage: 10^50 + 1. - 10^50
...: 0.000000000000000
```

Le problème ici est que le fait d'avoir entré 1. impose à SAGE de se placer dans les nombres réels qui sont toujours approchés (avec 53 bits de précision par défaut). Cette précision n'est pas assez grande pour notre calcul et SGAE ne "voit" plus le +1. On obtient 0. Si par contre on tape la commande suivante SAGE renvoie :

```
sage: 10^50 + 1 - 10^50
...: 1
```

SAGE a maintenant fait un calcul formel comme nous le faisons nous même et en simplifier les deux 10^{50}.

Un autre exemple : sage peut traiter des entiers aussi grand qu'on le veut. Les seules limites sont la places dans la machines et le temps.

Exemple 1.0.2 Voici quelques résultats de calcul de puissance de 2 avec SAGE :

```
sage: 2^100
...: 1267650600228229401496703205376

sage: 2^1000
...: 10715086071862673209484250490600018105614048117055336074437503883703510511249361224931983788156958581275946729175531468251871452856923140435984577574698574803934567774824230985421074605062371141877954182153046474983581941267398767559165543946077062914571196477686542167660429831652624386837205668069376
```

On que que même si le calcul est instantanné avec SAGE, c'est vite la place qui va manquer...

Comment gère-t-on les entiers de tailles arbitraires ? L'ordinateur en tant que machine ne peut traiter que des objets de tailles finie. En gén'eral, le processeur peux gérer des données de 32 ou de 64 bits. Si on transforme ces bits en entiers, on peut avoir des entiers de tailles maximale 2^{64} traités par l'ordinateur. Or on vient de voir que sage nous donne immédiatement les valeurs de 2^{100} ou même 2^{1000} (et même $2^{1000000}$ à partir duquel sage n'affiche plus toute la réponse). Par exemple pour $2^{10000000}$ si on utilise la commande `%\time` on obtient la reponse suivante de sage :

```
...: CPU time: 0.86 s, Wall time: 1.07 s
```

ce qui signifie que SAGE a mis 1,07 secondes pour faire la calcul (et 0,86 secondes pour le processeur.

Comment un ordinateur gère-t-il les grand entiers (bien plus grands que la taille de son processeur) ? Il procède exactement comme nous : il les découpe en morceaux (comme l'écriture en base 10) avec des chiffres. L'ordinateur va écrire en base 2^{32} ou 2^{64} selon la taille de son processeur (32 ou 64 bits). On va dans la suite se placer dans le cas de processeurs de 64 bits.

Définition 1.0.3 Entiers machine et multiprécision.

1. Un **entier machine** est un nombre entier que le processeur de la machine meut gérer directement. C'est donc un entier de taille 2^{64} (64 bits).

2. Un **entier multiprécision** a est un nombre entier (beaucoup) plus grand qu'un entier machine. On l'écrit en base 2^{64} sous la forme suivante :

$$a = (-1)^s \sum_{i=0}^{n} a_i 2^{64i},$$

où $s \in \{0; 1\}$ est le signe de l'entier a, n est un entier machine tel que $0 \leq n+1 < 2^{63}$, c'est la **taille** de a et les a_i sont des entiers machines. On code également la tailles et le signe de a dans les premiers bits. L'entier a sera donc une suite d'entiers machines :

$$a = (s', a_0, \cdots, a_n)$$

où $s' = s \cdot 2^{63} + n + 1$.

Remarque 1.0.4 Comme on a $0 \leq n+1 < 2^{63}$, la valeur de s' permet de retrouver à la fois celle de s et celle de n et donc le signe et la taille de notre entier. En effet, on a

— Si $s' = s \cdot 2^{63} + n + 1 < 2^{63}$ alors $s = 0$ et $n + 1 = s \cdot 2^{63} + n + 1 = s'$

— Si $s' = s \cdot 2^{63} + n + 1 geq 2^{63}$ alors $s = 1$ et $n + 1 = s \cdot 2^{63} + n + 1 - 2^{63} = s' - 2^{63}$

Exemple 1.0.5 L'entier 1 s'écrit $(1,1)$ c'est-à-dire $s'=1$ et donc $s=0$ et $n=0$. On a aussi $a_0=1$.

L'entier -1 s'écrit $(2^{63}+1,1)$ c'est-à-dire $s'=2^{63}+1$ et donc $s=1$ et $n=0$. On a aussi $a_0=1$.

Par convention l'entier 0 s'écrit (0).

Il est souvent très utile de connaitre la taille de l'entier à l'avance (d'où l'entier machine qui donne n). Comme on se limite pour n à un entier machine, ceci donne tout de même une limite aux entiers multiprécisions.

Proposition 1.0.6 Un entier multiprécision est compris dans l'intervalle

$$[-2^{64\cdot 2^{63}}+1, 2^{64\cdot 2^{63}}-1].$$

Preuve. Exercice. ∎

Pour stocker les plus grands entiers multiprécision, il faut tout de même 1^{11} GB (Gigas) ou encore 10^8 TB (Teras) ce qui représente plus de 1 millions de diques de 1 To chacun à plus de 500 euros donc un coup de plus 500 millions d'euros pour stocker de tels entiers (sans compter la place prise par ces disques de stockage...).

Remarque 1.0.7 On peut retrouver n à partir de a de la manière suivante

$$n = \lfloor \log_{2^{64}} |a| \rfloor - 1 = \left\lfloor \frac{\log_2 |a|}{64} \right\rfloor - 1.$$

Le nombre d'entiers machines nécessaires à décrire a est $\lambda(a) = n+2$.

2. Addition

2.1. Additionner les entiers multiprécisions

Dans cet paragraphe, nous allons donner un algorithme rudimentaire pour expliquer comment l'ordinateur gère l'additions d'entiers multiprécision. Pour celà, on suppose que notre processeur est capable d'additionner deux entiers machines. Plus précisément, on suppose que notre processeur dispose de l'opération suivante

$\texttt{ADDITION}(a, b) = (c, \gamma)$

Ici, les entrées a et b sont des entiers machines *i.e* $a, b \in [0, 2^{64} - 1]$ et la sortie (c, γ) est une paire formée d'un entier machine c d'un bit $\gamma \in \{0; 1\}$ (la retenue). La formule qui définit (c, γ) est la suivante :

$$a + b = \gamma \cdot 2^{64} + c.$$

Nous nous contenterons de donner un algorithme qui additionne des entiers multiprécisions sans signe et de la même taille. Nos entiers sont donc de la forme $a = (n + 1, a_0, \cdots, a_n)$ et $b = (n+1, b_0, \cdots, b_n)$.

Algorithme 2.1.1 (Addition des entiers multiprécisions)

$\gamma_0 := 0$
`For` i `in range` $(0, n + 1)$`:`
    (d_i, α_i) :=`ADDITION`(a_i, b_i)
    (c_i, β_i) :=`ADDITION`(d_i, γ_i)
    $(c_i, \gamma_{i+1}) := (c_i, \alpha_i + \beta_i)$
$c_{n+1} := \gamma_{n+1}$
`If` $c_{n+1} \neq 0$ `and` $n + 2 = 2^{63}$ `return` Somme trop grande !
`If` $c_{n+1} \neq 0$ `and` $n + 2 < 2^{63}$ `return` $(n + 2, c_0, \cdots, c_{n+1})$
`If` $c_{n+1} = 0$ `return` $(n + 1, c_0, \cdots, c_n)$

Exemple 2.1.2 Pour illustrer l'algorithme ci-dessus, nous allons faire une addition en base 10 (et pas 2^{64}) de deux entiers. On se propose de faire l'opération

$$8438 + 1945$$

avec $a = 8438 = (4; 8, 3, 4, 8)$ et $b = 1945 = (4; 5, 4, 9, 1)$ (le premier terme est la taille). On représente les étapes de l'algorithme dans un tableau

i	0	1	2	3	4
a_i	8	3	4	8	
b_i	5	4	9	1	
d_i	3	7	3	9	
α_i	1	0	1	0	
c_i	3	8	3	0	1
β_i	0	0	0	1	0
γ_i	0	1	0	1	1

On obtient

$$8438 + 1945 = 10383.$$

Exercice 2.1.3 Écrire un algorithme qui permet d'additionner des entiers multiprécisions quelconques (avec signes et de tailles différentes).

Proposition 2.1.4 L'algorithme d'addition des entiers multiprécisions de taille n et sans signe utilise exactement $2(n + 1)$ operations processeur `ADDITION`.

Preuve. On fait deux `ADDITIONS` pour chaque entier i entre 0 et n. ∎

Exercice 2.1.5

1. Écrire un programme qui fait passer un entier dans sa représentation en tant qu'entier multiprécision ainsi qu'on programme qui fait l'inverse.

2. Écrire un programme qui fait l'addition des entiers multiprécisions et tester s'il fonctionne grâce au programme précédent.

Simplification : Pour éviter d'avoir à tester des entiers trop grands, on pourra aussi réaliser les programmes ci-dessus avec des entiers machines de 1 bits (et donc on écrit les entiers en base 2).

Remarque 2.1.6 La retenue qu'on a géré avec les paramètres γ_i ci-dessus complique le calcul d'efficacité (ou de complexité) de cet algorithme. Pour évaluer la complexité, on va utiliser un modèle plus simple : les polynômes.

2.2. Addition des polynômes

À la place des entiers, on va maintenant considérer les polynômes en une variable x. Il peut sembler que ceci introduit de la complexité mais il n'en sera rien : on va grâce à cette astuce s'éviter toute retenue dans les opérations arithmétiques.

Définition 2.2.1 Un **polynôme de grand degré** sur un anneau A est la donnée

$$p(x) = \sum_{i=0}^{n} a_i x^i$$

où les a_i sont des éléments de A et n est un entier appelé **degré ou taille**. Pour la machine le polynôme est représenté par $(n, a_0, \cdots, a_n)$.

Remarque 2.2.2 Si on remplace x par 2^{64} et A par les entiers machines, on retrouve l'écriture des entiers multiprécision. L'arithmétique des (addition, soustraction, multiplication et division) entiers et des polynômes sont très semblable mais celle des polynômes est un peu plus simple : il n'y a jamais de retenue. On va donc utiliser cette dernière pour les calculs de complexité. Dans beaucoup de situations les calculs de complexité pour les entiers donnent des résultats semblables à ceux de polyômes.

L'opération `ADDITION` sera remplacer par l'addition dans l'anneau A. Encore une fois, on va supposer que nos polynômes ont le même degré. C'est un exercice facile que d'adapter l'algorithme suivant au cas général. On suppose donc que a et b sont de la forme $a = (n, a_0, \cdots, a_n)$ et $b = (n, b_0, \cdots, b_n)$.

Algorithme 2.2.3 (Addition des polynômes)

`For` i `in range` $(0, n+1)$`:`
    $c_i := a_i + b_i$
`return` $(n, c_0, \cdots, c_n)$

Proposition 2.2.4 L'algorithme d'addition des polynômes de taille n utilise exactement $n+1$ additions dans A.

Preuve. On fait une addition pour chaque entier i entre 0 et n. ∎

L'opération d'addition n'est donc pas une opération très coûteuse : de l'ordre de la taille des objets à additionner. L'opération importante en arithmétique et qui peut être très couteuse en temps est la multiplication.

2.3. Polynômes et sage

Pour travailler avec des polynômes dans sage, c'est assez simple. Pour définir une variable x ou y dans sage, on tape en général la commande :

```
sage: x = var('x') ; y = var('y')
```

On effectue alors des calculs sur des expressions formelles. Par exemple

```
sage: x = var('x') ; p = (2*x+1)*(x+2)*(x^4-1)
```

On dispose pour ces expressions une fonction degré :

```
sage: p.degree(x)
 ...: 6
```

mais sage ne change pas l'expression pour la représenter :

```
sage: p
 ...: (2*x+1)*(x+2)*(x^4-1)
```

Sage permet de traiter les polynômes de façon plus algébrique et calculer comme dans l'anneau des polynômes $R = A[x]$ par exemple avec $A = \mathbb{Q}$, $A = \mathbb{R}$ ou encore $A = \mathbb{Z}/4\mathbb{Z}$. Pour les polynômes à coefficients dans $\mathbb{Q}$ (qui est représenté par `QQ` dans sage). On tape :

```
sage: x = polygen(QQ,'x')
```

Si maintenant on considère le même polynôme, sage dévelope automatiquement le polynôme. La fonction degré fonctionne toujours :

```
sage: p = (2*x+1)*(x+2)*(x^4-1) ; p ; p.degree()
 ...: 2*x^6 + 5*x^5 + 2*x^4 - 2*x^2 - 5*x - 2
 ...: 6
```

Pour construire l'anneau $R = \mathbb{Q}[x]$, on tape :

```
sage: R = PolynomialRing(QQ,'x') ; R
 ...: Univariate Polynomial Ring in x over Rational Field
```

3. Multiplication

Comme on vient de le voir. Les algorithmes et le calcul de la complexité sont souvnt plus facile avec les polynômes qu'avec les entiers multiprécisions même s'ils sont de nature semblable. Nous allons donc commencer par nous intéresser au cas de la multiplication des polynômes.

Nous donnons ici l'algorithme naif de multiplication de deux polynômes $a = \sum_{i=0}^{n} a_i x^i$ et $b = \sum_{i=0}^{m} b_i x^i$ avec $m \leq n$

Algorithme 3.0.1 (Algorithme "naif" de multiplication des polynômes)

`For` k `in range` $(0, n+m+1)$:
    $c_k := 0$
    `For` i `in range` $(\max(0, k-m), \min(k, n)+1)$:
        $c_k = c_k + a_i b_{k-i}$

`Return` $\sum_{k=0}^{n+m} c_k x^k$.

Proposition 3.0.2 Pour multiplier deux polynômes de degrés n et m, l'algorithme naïf nécessite $(m+1)(n+1)$ additions et $(m+1)(n+1)$ multiplications.

Preuve. On effectue deux boucles et pour chaque boucle on doit faire une addition et une multiplication. On doit donc effectuer la somme suivante

$$S = \sum_{k=0}^{n+m} \sum_{i=\max(0,k-m)}^{\min(k,n)} 1.$$

On coupe la première somme en trois afin de calculer cette double somme correctement. Pour celà on distingue trois cas : $k \in [0, m-1]$ ou $k \in [m, n]$ ou $k \in [n, n+m]$. Dans le premier cas on calcule la somme

$$S_1 = \sum_{k=0}^{m-1} \sum_{i=0}^{k} 1 = \sum_{k=0}^{m-1} (k+1) = \frac{m(m+1)}{2}.$$

Dans le second cas

$$S_2 = \sum_{k=m}^{n} \sum_{i=k-m}^{k} 1 = \sum_{k=m}^{n} (m+1) = (m+1)(n-m+1).$$

Dans le dernier cas

$$S_3 = \sum_{k=n+1}^{n+m} \sum_{i=k-m}^{n} 1 = \sum_{k=n+1}^{n+m} (n+m+1-k) = m(n+m+1) - \frac{(n+m+1)(n+m)}{2} + \frac{n(n+1)}{2}.$$

On obtient $S = S_1 + S_2 + S_3 = (m+1)(n+1)$. ∎

On peut sans trop réfléchir faire un peu mieux. Pour celà on remarque que faire un produit d'un scalaire a_k par un polynôme $b = \sum_{i=0}^{m} b_i x^i$ nécessite $m+1$ multiplications :

$$a_k \cdot b = \sum_{i=0}^{m} a_k b_i x^i.$$

On peut alors proposer un nouvel algorithme.

Algorithme 3.0.3 (Algorithme "naif amélioré")

`For` k `in range` $(0, n+1)$`:`
    $d_k := (a_k \cdot b)x^k$

`Return` $\sum_{k=0}^{n} d_k$.

Proposition 3.0.4 Pour multiplier deux polynômes de degrés n et m, l'algorithme naïf amélioré nécessite au plus mn additions et $(m+1)(n+1)$ multiplications.

Preuve. À chaque tour de boucle on fait une opération $a_k \cdot b$ qui nécessite $m+1$ multiplications. La multiplication par x^i ne coute rien, c'est juste un décalage dans les coefficients. On a donc $(n+1)(m+1)$ multiplications. On a ensuite n additions de polynômes. En regardant de près, on voit qu'à chaque étape, on ne doit faire que m additions pour sommer ces polynômes car ils sont "décalés :

$$\begin{array}{ccccccccc} a_{k+1}b_m x^{m+k+1} & + & a_{k+1}b_{m-1}x^{m+k} & + & \cdots & + & a_{k+1}b_0 x^{k+1} & & \\ & & a_k b_m x^{m+k} & + & \cdots & + & a_k b_1 x^{k+1} & + & a_k b_0 x^k \end{array}$$

ce qui donne nm additions. ∎

Corollaire 3.0.5 Les algorithmes naifs permettent de multipler des polynômes de degrés n et m en au plus $2nm + n + m + 1$ opérations soit en $O(nm)$.

L'avantage du second algorithme est de permettre d'écrire un algorithme pour les entiers sans avoir à prendre en compte les retenues. On écrit nos entiers sous la forme $a = \sum_{i=0}^{n} a_i 2^{64i}$ et $b = \sum_{i=0}^{m} b_i 2^{64i}$ avec $m \leq n$. On commence par définir

$$a_k \cdot b = \sum_{i=1}^{m} a_k b_i 2^{64i}.$$

Ici on aurait besoin d'être plus précis et de prendre en compte les retenues : le produit de deux entiers machines $a_k b_i$ peut dépasser la taille des entiers machine.

Algorithme 3.0.6 (Algorithme "naif" de multiplication des entiers)

`For` k `in range` $(0, n+1)$:
$d_k := (a_k \cdot b)2^{64k}$

`Return` $\sum_{k=0}^{n} d_k$.

Proposition 3.0.7 Pour multiplier deux entiers multiprécisions de tailles n et m, l'algorithme naïf amélioré nécessite au plus $O(mn)$ opérations processeur.

Exercice 3.0.8 Donner un algorithme qui permette d'écrire l'opération $a_k \cdot b$ ci-dessus et évaler la complexité de l'algorithme naif utilisant cet algorithme.

Remarque 3.0.9 On voit que la multiplication est beaucoup plus "coûteuse" que l'addition. Il va donc falloir chercher à minimiser le nombre de multiplications dans les algorithmes ainsi qu'améliorer l'algorithme de multiplication. On vera des algorithmes plus efficaces pour la multiplication par la suite, notamment l'algorithme de Karatsuba et la transformation de Fourier discrète.

4. Division euclidienne

La division euclidienne outre son intérêt intrinsèque est très utile dans de nombreuses situations :

- test d'exactitude : pour vérifier que deux entiers a et b sont égaux, on vérifie que $a \equiv b \pmod{n}$.
- en cryptographie (code RSA par exemple)
- pour l'acceleration des algorithmes de multiplication.

Remarque 4.0.1 Pour les entiers, la division euclidienne est bien définie. Par contre, pour les polynômes $A[x]$ à coefficients dans un anneau A, la division euclidienne n'est pas bien définie. On a alors deux alternatives :

- choisir pour A un corps, la division euclidienne existe alors toujours,
- ne faire que des division par des polynômes dont le coefficient dominant est inversible.

C'est cette dernière solution plus générale que nous allons choisir. Soient donc $a = \sum_{i=0}^{n} a_i x^i$ et $b = \sum_{i=0}^{m} b_i x^i$ deux polynômes tels que b_m soit inversible. On notera $\mathrm{LC}(r)$ le coefficient dominant d'un polynôme r.

Algorithme 4.0.2 (Division euclidienne des polynômes)

$r = a$; $u = \mathrm{LC}(b)^{-1}$; $q = 0$
`For` i `from` $n - m$ `to` 0`:`
    `If` $\deg(r) == m + i$:
        $q = q + \mathrm{LC}(r) u x^i$; $r = r - \mathrm{LC}(r) u x^i b$
`return` (q, r)

Exemple 4.0.3 Soit $a = 3x^4 + 2x^3 + x + 5$ et $b = x^2 + 2x + 3$. On pose la division

$$
\begin{array}{rrrrrrrrrrrrr|lllll}
 & 3x^4 & + & 2x^3 & + & \cdot & + & x & + & 5 & x^2 & + & 2x & + & 3 \\
\cline{14-18}
- & (3x^4 & + & 6x^3 & + & 9x^2) & & & & & 3x^2 & - & 4x & - & 1 \\
\cline{1-10}
 & & & -4x^3 & - & 9x^2 & + & x & + & 5 & & & & & \\
 & & - & (-4x^3 & - & 8x^2 & - & 12x) & & & & & & & \\
\cline{3-12}
 & & & & & -x^2 & + & 13x & + & 5 & & & & & \\
 & & & & - & (-x^2 & - & 2x & - & 3) & & & & & \\
\cline{5-12}
 & & & & & & & 15x & + & 8 & & & & &
\end{array}
$$

pour obtenir $q = 3x^2 - 4x - 1$ et $r = 15x + 8$.

Proposition 4.0.4 Cet algorithme fournit une paire (q, r) de polynômes qui vérifient $a = bq + r$ avec $\deg(r) < \deg(b)$.

Preuve. On voit qu'à chaque étape, on a l'égalité suivante : $a = bq + r$. En effet, au départ c'est le cas : $q = 0$ et $r = a$. Ensuite, si $\deg(r) = m + i$, on a $bq + r = b(q - \mathrm{LC}(r)ux^i) + (r - \mathrm{LC}(r)ux^i b)$. Sinon, on ne change rien.

De plus, on voit que, si à un moment r est de degré supérieur ou égal à $n - m$, on va remplacer r par $r - \mathrm{LC}(r)ux^i b$ ce qui supprime le terme dominant de r et fait diminuer son degré. En fin d'algorithme, le degré de r est donc strictement inférieur à $m = \deg(b)$.

Reste à montrer l'unicité. Pour celà soient $a = bq + r = bq' + r'$ deux écritures avec $\deg(r), \deg(r') < \deg(b)$. On a $r - r' = b(q' - q)$ et donc b divise $r - r'$. Comme $\deg(r - r') < \deg(b)$ ceci n'est possible que si $r - r' = 0$ donc $r = r'$ et $q = q'$. ∎

Proposition 4.0.5 Cet algorithme utilise $(n - m + 1)(m + 2)$ multiplications et $(n - m + 1)m$ additions soit en tout $(n - m + 1)(2m + 1)$ opérations dans A.

Preuve. À chaque cycle, on multiplie $\mathrm{LC}(r)$ avec u, soit une multiplication, puis le produit $\mathrm{LC}(r)u$ avec b, soit $m + 1$ multiplications.

La soustraction correspond à additionner un polynôme avec $m + 1$ entrées et donc $m + 1$ additions. En fait, on a vu que le terme dominant de r devient nul donc on a pas besoin d'effectuer la première soustraction. Donc m additions. L'addition sur q n'ajoute qu'un monôme à un polynôme qui n'avait pas encore ce monôme, on a donc ici aucune addition.

Comme on fait $n - m + 1$ boucles, on obtient le résultat. ∎

Rapellons rapidement la division euclidienne dans le cas des entiers positifs et donnons en un algorithme. Soient donc a et b deux entiers avec $b \neq 0$.

Algorithme 4.0.6 (Division euclidienne)

$r = a$; $q = 0$
`While` $r \geq b$:
    $r = r - b$; $q = q + 1$
`return` (q, r)

Proposition 4.0.7 Cet algorithme fournit une unique paire (q, r) d'entiers positifs tels que $a = bq + r$ et $r < b$.

Preuve. À chaque étape, on a $bq + r = b(q + 1) + (r - b)$ qui reste donc constant tout au long de l'algorithme. Comme au départ, on a $q = 0$ et $r = a$, on a bien $a = bq + r$. L'algorithme se termine car r est décroissante strictement et sera donc à un moment inférieur strictement à b. L'unicité se prouve comme pour les polynômes : si on a $a = bq + r = bq' + r''$ avec $r, r' < b$, alors $r - r' = b(q - q')$. Mais on doit avoir $|r - r'| < b$ donc $r - r' = 0$ et $r = r'$ et $q = q'$. ∎

Proposition 4.0.8 Cet algorithme effectue exactement $q + 1$ additions où q est le quotient de a par b.

Preuve. À chaque étape on effectue la soustraction $r = r - b$ (donc une addition) et l'opération $q = q + 1$ qui constiste à prendre le suivant (et ne compte pas comme addition). On effectue ces opérations jusqu'à ce que q soit égal au quotient de a par b d'où le résultat. ∎

Remarque 4.0.9 Un avantage de cet algorithme est qu'il ne nécessite aucune multiplication.

5. Polynômes creux

Pour les polynômes (ou les entiers multiprécision, il peut être plus économique de représenter les polynômes d'une autre manière appelée **représentation creuse**. Cette représentation consiste à donner, pour le polynôme $a = \sum_i a_i x^i$, les paires (i, a_i) telles que $a_i \neq 0$ et seulement ces paires.

Exemple 5.0.1 Pour le polynôme $a = x^{16} + 2$, la notation "classique" consiste à donner la liste des coefficients a_i en commançant par les unités, soit ici :

$$a = (2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1).$$

La représentation creuse est beaucoup plus simple (et c'est celle que nous utilisons en écrivant $x^{16} + 2$) :

$$a = ((0, 2), (16, 1)).$$

Exemple 5.0.2 Il est à noter que c'est ainsi que sage représente la décomposition en facteurs premiers d'un entier. Il ne donne pas la liste de tous les facteurs premiers mais seulement ceux qui interviennent effectivement. Par exemple, sage revoie pour la décomposition de 242 :

$$((2, 1), (11, 2))$$

ce qui signifie que le nombre premier 2 apparaît avec multiplicité 1 alors que 11 apparaît avec multiplicité 2, soit $242 = 2 \times 11^2$. Il ne donne pas la liste de tous les premiers jusqu'à 11 :

$$242 = 2^1 \times 3^0 \times 5^0 \times 7^0 \times 11^2.$$

Définition 5.0.3 Le **poids** d'un polynôme a est

$$\omega(a) = \big|\{i \in \mathbb{N} \mid a_i \neq 0\}\big|.$$

Remarque 5.0.4 On a toujours $\omega(a) \leq \deg(a)$.

Définition 5.0.5 Un polynôme est dit **creux** si son poids est "faible" (notion à définir en fonction des besoins).

On peut montrer la proposition suivante.

Proposition 5.0.6 Il existe un algorithme permettant de multiplier des polynômes a et b en au plus $2\omega(a)\omega(b) + \omega(a) + \omega(b) + 1$ opérations

Exercice 5.0.7 Proposer un algorithme qui réalise ces bornes.