Coût de l’algorithme d’Euclide et CAPES interne 2000

Dany-Jack Mercier(*)

Résumé : Voici quelques réflexions menées à partir d’un énoncé de CAPES interne qui proposait de majorer le nombre de divisions euclidiennes nécessaires à l’algorithme d’Euclide. On définit le coût d’un algorithme dans deux modèles différents (coûts fixes ou bilinéaires) pour mieux s’adapter aux méthodes de calcul de l’ordinateur, puis l’on exprime une majoration du coût de l’algorithme d’Euclide et de son cousin l’algorithme d’Euclide étendu. Une dernière partie étudie l’algorithme d’écriture d’un nombre en base *b*. Ce travail intéressera les candidats au CAPES, et sans doute aussi les agrégatifs pour la nouvelle épreuve de modélisation de l’agrégation externe.

1. Introduction

La première composition du CAPES interne 2000 montre une certaine rupture avec les énoncés précédents. S’il est maintenant facile de prédire que les probabilités risquent de former l’une des parties des CAPES à venir (cela a déjà été le cas dans les sessions 1999 et 2000), on peut s’étonner de trouver deux « nouveaux » thèmes de travail dans la session 2000. Sur les trois parties indépendantes formant l’énoncé de cette session, la première propose une analyse du temps de calcul de l’algorithme d’Euclide, et la seconde un problème de statistique prolongé par une étude de variables aléatoires et des calculs probabilistes associés. Cette tendance à l’algorithmique et aux statistiques pourrait se confirmer dans l’avenir compte tenu de l’introduction des nouveaux programmes de Lycée en 2002 (à titre indicatif, l’une des deux parties du CAPES interne 2002 demandait de construire un algorithme et de finaliser le programme sur sa calculatrice).

Nous nous proposons ici d’utiliser l’exercice 1 du CAPES interne 2000 comme tremplin pour préciser la notion de complexité d’un algorithme. Cette complexité est liée au temps de calcul, donc au nombre d’opérations algébriques à effectuer, et c’est une majoration du nombre de divisions euclidiennes nécessaires à l’algorithme d’Euclide qui est demandée dans l’exercice. Mais ce temps de calcul dépend aussi de la structure interne de l’ordinateur et de l’implémentation des nombres en base 2. Cela nous mène à préciser comment l’estimation peut être affinée en introduisant le modèle des coûts bilinéaires.

2. Un extrait du CAPES interne 2000

2.1. L’extrait (Exercice n°1)

Le calcul du plus grand commun diviseur $a \wedge b$ de deux nombres a et b ($a > b > 0$) par l’algorithme d’Eudide se fait par divisions successives comme dans l’exemple

(*) IUFM de Guadeloupe, Morne Ferret, BP 399, Pointe-à-Pitre cedex 97159, dany-jack.mercier©univ-ag.fr

suivant où l'on a choisi $a = 44$ et $b = 18$:

$$44 = 2 \times 18 + 8$$

$$18 = 2 \times 8 + 2$$

$$8 = 4 \times 2 + 0$$

et où l'on déduit $44 \wedge 18 = 2$. Si on désigne par $l(a,b)$ la longueur de l'algorithme, c'est-à-dire le nombre de divisions nécessaires pour aboutir au résultat, nous avons ici : $l(44,18) = 3$.

L'objet de l'exercice est de majorer $l(a,b)$. Pour cela on note $r_1, r_2, ..., r_n$ les restes des n divisions successives de l'algorithme (on a donc $l(a,b) = n$ et $r_n = 0$), et l'on pose commodément $a = r_{-1}$ et $b = r_0$.

1) Montrer que pour $k \in \{1, \ldots, n-1\}$, on a $r_{k-2} \geq r_{k-1} + r_k$.

2) Soit (u_n) la suite de Fibonacci définie par : $u_0 = u_1 = 1$ et $u_n = u_{n-2} + u_{n-1}$ pour $n \geq 2$. Montrer qu'en posant $\alpha = \dfrac{1+\sqrt{5}}{2}$ on a $\alpha^{n-1} \leq u_n$ pour tout $n \geq 1$.

3) Montrer que $r_{n-k-1} \geq u_k$ pour tout $k \in \{0, \ldots, n\}$, et en déduire que n vérifie une majoration de la forme $n \leq \mathrm{A} \ln a + \mathrm{B}$. Vérifier cette majoration sur un exemple (on pourra prendre $\mathrm{B} = 1$ et $\mathrm{A} < 2{,}1$).

2.2. Une solution

1) Les divisions euclidiennes successives peuvent s'écrire :

$$\textbf{(AE)} \quad \begin{cases} a = bq_1 + r_1 & \text{avec } 0 \leq r_1 < b, \\ b = r_1 q_2 + r_2 & \text{avec } 0 \leq r_2 < r_1, \\ r_1 = r_2 q_3 + r_3 & \text{avec } 0 \leq r_3 < r_2, \\ \cdots\cdots\cdots & \\ r_{k-2} = r_{k-1} q_k + r_k & \text{avec } 0 \leq r_k < r_{k-1}, \\ \cdots\cdots\cdots & \\ r_{n-3} = r_{n-2} q_{n-1} + r_{n-1} & \text{avec } 0 \leq r_{n-1} < r_{n-2}, \\ r_{n-2} = r_{n-1} q_n + r_n & \text{avec } r_n = 0. \end{cases}$$

Si $1 \leq k \leq n-1$, aucun des restes r_k n'est nul, et donc $q_k \geq 1$ (en effet, les dividendes et diviseurs qui interviennent ici sont positifs, donc $q_k \geq 0$, et l'égalité $q_k = 0$ entraîne $r_{k-2} = r_k$, ce qui est absurde puisque la suite $(r_k)_{1 \leq k \leq n}$ est strictement décroissante). Par conséquent

$$r_{k-2} = r_{k-1} q_k + r_k \geq r_{k-1} + r_k.$$

2) On a clairement $\alpha^{n-1} \leq u_n$ pour $n = 1$ ou 2. Si cette inégalité est vraie jusqu'au rang $n-1$, on obtient :

$$u_n = u_{n-2} + u_{n-1} \geq \alpha^{n-3} + \alpha^{n-2} = \alpha^{n-3}(1+\alpha) = \alpha^{n-3} \times \alpha^2 = \alpha^{n-1}.$$

au rang $n \geq 3$, puisque le nombre d'or α vérifie $\alpha^2 = \alpha + 1$. L'inégalité est démontrée par récurrence.

3) Montrons que $r_{n-k-1} \geq u_k$ par récurrence sur $k \in \{0, \ldots, n\}$. C'est vrai pour $k = 0$ puisque $r_{n-1} \geq u_0 = 1$. C'est vrai pour $k = 1$ puisque $r_{n-2} > r_{n-1} \geq u_1 = 1$. Si l'inégalité est vraie jusqu'au rang $k - 1 < n$, alors $r_{n-k-1} \geq r_{n-k} + r_{n-k-1} \geq u_{k-1} + u_{k-2} = u_k$ et la récurrence aboutit. La question 2) montre alors que $r_{n-k-1} \geq u_k \geq \alpha^{k-1}$, d'où

$$\forall\, k \in \{0, \ldots, n\}\ r_{n-k-1} \geq \alpha^{k-1}.$$

En faisant $k = n$ on obtient :

$$r_{-1} = a \geq \alpha^{n-1} \Rightarrow \ln a \geq (n-1)\ln\alpha \Rightarrow n \leq \frac{\ln a}{\ln\alpha} + 1 \leq \mathrm{A}\ln a + \mathrm{B}$$

avec $\mathrm{A} = \dfrac{1}{\ln\alpha} \cong 2{,}07 < 2{,}1$ et B = 1. L'exemple de l'énoncé donne $l(44,18) = 3$ pour une majoration théorique $n \leq 2{,}07 \times \ln 44 + 1 \cong 8{,}83$.

2.3. Quelques remarques

1) 11 est remarquable que l'énoncé du CAPES évite de demander une justification théorique de l'algorithme d'Euclide **(AE)**. L'algorithme se termine dès que l'un des restes obtenus est nul, et c'est toujours le cas, autrement la suite (r_k) des restes serait strictement décroissante dans **N**. Par ailleurs, les égalités

$$\mathrm{pgcd}\,(a,b) = \mathrm{pgcd}\,(b,r_1) = \ldots = \mathrm{pgcd}\,(r_{n-1},0) = r_{n-1}$$

montrent bien que le pgcd de a et b coïncide avec le dernier reste non nul r_{n-1} obtenu.

2) La première majoration de n est attribuée au mathématicien français Gabriel Lamé. Dans un article paru en 1844, Lamé démontre que le nombre n de divisions nécessaires pour écrire l'algorithme du pgcd de deux entiers a et b avec $a \geq b$ est majoré par « 5 fois le nombre de chiffres décimaux de b » ([5], p. 403). Cela s'écrit $n \leq 5\,([\log_{10} b] + 1)$ et entraîne $n \leq \dfrac{5}{\ln 10}\ln a + 5$ où $\dfrac{5}{\ln 10} \cong 2{,}17$. On devine maintenant pourquoi l'énoncé du CAPES précise que la constante A est inférieure à 2,1.

3) Un majorant du temps de calcul de l'algorithme d'Euclide avait déjà été donné par Antoine-André-Louis Reynaud en 1811, et les résultats de Lamé étaient connus d'Émile Léger en 1837, et rigoureusement démontrés par Pierre-Joseph-Étienne Finck en 1841 ([5]). À l'époque, le calcul de l'efficacité d'un algorithme était considéré comme une simple curiosité mathématique, mais les temps ont vraiment changé avec l'apparition des machines. Quoiqu'il en soit, l'algorithme d'Euclide n'a pas pris une ride depuis 23 siècles d'existence et l'on peut dire, avec D. E. Knuth, qu'il s'agit du « plus ancien algorithme non trivial qui ait survécu à ce jour ».

4) Dans l'article [1] initiateur de la cryptographie RSA (sur RSA : [4}), les auteurs citent la majoration $n \leq [2 \log_2 a]$ pour justifier la faisabilité du système, un système de chiffrement embarqué sur des cartes bancaires ne devant pas entraîner des queues interminables aux caisses des supermarchés ! L'importance du temps de calcul est bien une donnée récente…

Il est facile de vérifier que la majoration donnée en [1] est une conséquence de celle de l'énoncé du CAPES lorsque $a \geq 4$. On démontrera plus loin la majoration $n \leq \frac{3}{2}\log_2 a + 1$. De toute façon, d'un point de vue moderne, l'important n'est pas tant le coefficient devant le logarithme, mais le résultat asymptotique permettant d'affirmer que le nombre n de divisions est dominé par $\log_2 a$, ce que l'on écrit $n = \mathrm{O}(\log_2 \mathrm{a})$.

5) Toujours d'un point de vue récent, la majoration du nombre d'opérations algébriques nécessaires pour accomplir un algorithme ne suffit pas à rendre compte du travail sur machine, et l'on est obligé d'avoir recours à des modèles plus précis. C'est l'objet de la Section suivante.

3. Le modèle des coûts bilinéaires

3.1. Deux hypothèses possibles

Pour comparer deux algorithmes arithmétiques entre eux, on essaie de trouver un ordre de grandeur – ou un majorant de cet ordre de grandeur – du temps écoulé entre le moment où l'on démarre le programme et celui où il fournit la solution. II est certain que le nombre d'opérations en jeu influe sur le temps d'exécution, et en ce sens, le résultat de Lamé clôt le débat. Mais, dans la pratique, la machine n'effectuera pas toutes les opérations avec un même coût, et il est nécessaire d'affiner l'analyse. On envisage donc les points de vue suivants ([3], §1.3).

Hypothèse des coûts fixes

Dans ce modèle, les opérations arithmétiques d'addition, de soustraction, de multiplication et de division, sont fournies avec la machine et possèdent un coût indépendant de la taille des facteurs. Le coût de a additions ou soustractions, de m multiplications et de d divisions est de la forme $a\mathrm{A} + m\mathrm{M} + d\mathrm{D}$ où A, M, D sont des constantes. Cela signifie que seul le nombre d'opérations est réellement pris en compte. Si cette hypothèse est acceptable lorsqu'on manipule de « petits » nombres, elle devient vite irréaliste lorsque les entiers atteignent des tailles importantes.

Hypothèse des coûts bilinéaires

On suppose ici que les entiers sont de taille trop grande pour que le coût des opérations arithmétiques soit constant. Étant dans la nécessité de manipuler des nombres « par tranches » en utilisant leurs représentations binaires et en reproduisant des techniques opératoires classiques, on constate que chacune des quatre opérations possède un coût qui dépend essentiellement de la « taille » des facteurs.

Dans toute la suite, on se placera dans l'hypothèse des coûts bilinéaires qui traduit mieux la réalité calculatoire, et il nous faut maintenant préciser la notion de « taille » d'un entier, puis calculer les « prix de revient » – au sens algorithmique – des quatre opérations standard de $\mathbf{Z}$ (Section 3.2). C'est seulement après ce travail que nous pourrons évaluer le coût de l'algorithme d'Euclide.

3.2. Coût des opérations dans le modèle à coûts bilinéaires

Notons $\log_2$ la fonction logarithme en base 2 et $[x]$ la partie entière du réel x. Soit m un entier naturel non nul. L'écriture de m en base 2 utilise $k + 1$ chiffres si et seulement si il existe $k + 1$ bits $a_0, a_1, \ldots, a_{k-1}$ (on rappelle qu'un bit est un 0 ou un 1) tels que

$$m = 2^k + a_{k-1}2^{k-1} + \ldots + a_1 \times 2 + a_0.$$

Cela équivaut à $2^k \leq m < 2^{k+1}$, autrement dit à $k \leq \log_2 m < k + 1$. Ainsi il faut exactement $[\log_2 m] + 1$ chiffres pour écrire un entier m en base 2, et nous dirons commodément que la taille d'un entier m est égale à $[\log_2 m]$.

Considérons l'addition (ou la soustraction) de deux entiers naturels m et n. Ces entiers sont connus par leurs écritures binaires qui utilisent chacune $[\log_2 m] + 1$ et $[\log_2 n] + 1$ chiffres. L'addition s'effectue en calculant

$$\kappa(m,n) := \sup\left(([\log_2 m] + 1), ([\log_2 n] + 1)\right)$$

additions bit à bit. Le coût de ces opérations élémentaires – c'est-à-dire la durée de leur exécution – est majoré par un nombre proportionnel à $\kappa(m,n)$, c'est-à-dire de la forme $c \times \kappa(m,n)$ où le coefficient de proportionnalité c dépend des capacités techniques du système informatique employé. La constante c est directement liée au nombre d'opérations bit à bit que la machine peut accomplir en une seconde.
L'existence d'une constante c qui dépend du système informatique utilisé explique pourquoi on évalue le coût d'un algorithme en donnant seulement une majoration du temps passé par le programme pour arriver au résultat, et cela pour des entrées arbitrairement grandes. Cette majoration reste proportionnelle au nombre d'opérations effectuées bit à bit.
Comme

$$\frac{\kappa(m,n)}{\log_2 m} \leq \frac{\log_2 m + 1}{\log_2 m} \leq 2$$

dès que $2 \leq n \leq m$, il existera une constante c_+ et un entier N tels que le coût de l'algorithme de l'addition de m et n soit majoré par $c_+ \times \sup(\log_2 m, \log_2 n)$ pour tous n et m supérieurs ou égaux à N. Dans ce cas, on dit que l'algorithme d'addition de m et n est de complexité $O(\sup(\log_2 m, \log_2 n))$, que son coût est en $\sup(\log_2 m, \log_2 n)$, ou encore que son coût est majoré par $c_+ \sup(\log_2 m, \log_2 n)$.

Considérons maintenant la multiplication de deux entiers naturels m et n. Les écritures binaires de m et n utilisent chacune $[\log_2 m] + 1$ et $[\log_2 n] + 1$ chiffres, et le produit s'effectue bit à bit en reproduisant la disposition habituelle de la multiplication.
On dénombre $([\log_2 m] + 1)([\log_2 n] + 1)$ multiplications bit à bit, suivies de $[\log_2 n]$ sommes d'entiers possédant moins de $([\log_2 m] + 1) + [\log_2 n])$ chiffres binaires. Au total, la multiplication nécessite moins de

$$\xi(m,n) := ([\log_2 m] + 1)([\log_2 n] + 1) + ([\log_2 m] + [\log_2 n] + 1)[\log_2 n]$$

opérations bit à bit. Si l'on suppose $2 \leq n \leq m$, on obtient

$$\begin{aligned}\xi(m,n) &\leq (\log_2 m+1)(\log_2 n+1)+(2\log_2 m+1)(\log_2 n)\\ &\leq \left[\left(1+\frac{1}{\log_2 m}\right)\left(1+\frac{1}{\log_2 n}\right)+2+\frac{1}{\log_2 m}\right](\log_2 m)(\log_2 n)\\ &\leq 7(\log_2 m)(\log_2 n).\end{aligned}$$

Cela montre que la multiplication de deux entiers naturels m et n a un coût majoré par $c_\times$ ($\log_2 m$) ($\log_2 n$), où $c_\times$ est une constante qui dépend du nombre d'opérations bit à bit que la machine peut accomplir en une seconde. L'algorithme de multiplication est de complexité O (($\log_2 m$) ($\log_2 n$)).

Le travail est identique lorsqu'on s'intéresse à la division de m par n (avec $1 \leq n \leq m$). Écrivons m et n en binaire, puis posons la division de façon usuelle. Le dividende m compte $[\log_2 m] + 1$ chiffres binaires, le diviseur en compte $[\log_2 n] + 1$, et il est facile de voir que le quotient en compte $[\log_2 m] - [\log_2 n] + 1$. Chacun des chiffres du quotient est multiplié par chacun des $[\log_2 n] + 1$ chiffres du diviseur, puis donne lieu à la soustraction entre deux nombres de moins de $[\log_2 m] + 1$ chiffres pour obtenir un reste partiel. Au total, on dénombre moins de

$$\zeta(m,n) := ([\log_2 n] + [\log_2 m] + 2)\ ([\log_2 m] - [\log_2 n] + 1)$$

opérations bit à bit. Alors

$$\begin{aligned}\zeta(m,n) &\leq 4[\log_2 m]([\log_2 m]-[\log_2 n]+1)\\ &\leq 4(\log_2 m)(\log_2 m-\log_2 n+2)\\ &\leq 8(\log_2 m)(1+\log_2 m-\log_2 n)\end{aligned}$$

et le coût de la division euclidienne est majoré par $c_\div$ ($\log_2 m$) ($1 + \log_2 m - \log_2 n$) où $c_\div$ est une constante.

Le tableau ci-dessous résume les coûts obtenus dans le modèle bilinéaire :

Opérations dans N :	**Coût majoré par :**
Addition ou soustraction $m \pm n$	c_+ sup ($\log_2 m$, $\log_2 n$)
Multiplication $m \times n$	$c_\times$ ($\log_2 m$) ($\log_2 n$)
Division de m par n (avec $1 \leq n \leq m$)	$c_\div$ ($\log_2 m$) ($1 + \log_2 m - \log_2 n$)(*)

(*) **Remarque** : En fait, le modèle retient parfois la valeur $c_\div(\log_2 m)(\log_2 m - \log_2 n)$ pour le coût d'une division. Ce choix n'est pas réaliste lorsque $m < 2n$. Dans ce cas, la division de m par n est triviale : le quotient vaut 1 et le reste vaut $m - n$; et le nombre d'opérations effectuées bit à bit est celui de la différence $m - n$, soit $[\log_2 m] + 1$, majoré par $2(\log_2 m)$. Le coût réaliste de la division est alors en $c_\div(\log_2 m)$ ou, à la rigueur, en $c_\div$ ($\log_2 m$) ($1 + \log_2 m - \log_2 n$) mais ne peut en aucun cas être dominé par l'expression ($\log_2 m$) ($\log_2 m - \log_2 n$) pour m et n arbitrairement grands (puisque cette dernière expression s'annule lorsque $m = n$).

Heureusement, si $m \geq 2n$, alors $1 \leq \log_2 m - \log_2 n$ et les inégalités précédentes donnent $\zeta(m,n) \leq 16$ ($\log_2 m$) ($\log_2 m - \log_2 n$), ce qui justifie un coût en ($\log_2 m$) ($\log_2 m - \log_2 n$).

4. Algorithme d'Euclide

4.1. Principe

L'algorithme d'Euclide (décrit à la Section 2) peut donner lieu à cette procédure Maple (a et b désignent des entiers relatifs quelconques et on utilise le quotient entier iquo) :

(AE) : Algorithme d'Euclide
Entrées : deux entiers relatifs quelconques a et b.
Sorties: d = pgcd(a,b).

```
>pgcd:=proc(a,b);
>u:=a ; v:=b;
>if v=0 then u fi;
>while v<>0 do w:=u ; u:=v; v:=w−(iquo(w,v))*v ; od;
>u;
>end;
```

4.2. Longueur

Pour a et b donnés (avec $a > b > 0$), notons $l(a,b)$ le nombre de divisions figurant dans l'algorithme d'Euclide. Avec les notations précédentes, $l(a,b) = n$. De façon intuitive, le nombre n de divisions à effectuer sera maximum lorsque chacun des quotients vaut 1, autrement dit lorsque $a = b + r_1$, puis $b = r_1 + r_2$, etc. La suite des restes vérifie alors la relation de récurrence $r_{i+2} = r_{i+1} + r_i$.
Ainsi la suite de Fibonacci (F_i), définie par $\mathrm{F}_0 = 0$, $\mathrm{F}_1 = 1$ et $\mathrm{F}_{i+2} = \mathrm{F}_{i+1} + \mathrm{F}_i$ pour tout entier i, fournit une infinité de couples (a,b) pour lesquels le calcul du pgcd utilise beaucoup de divisions. Par exemple,

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8
0	1	1	2	3	5	8	13	21

et le calcul de pgcd (21,13) demande 7 divisions : $21 = 13 + 8$, $13 = 8 + 5$, …, $2 = 1 + 1$, $1 = 1 + 0$. De façon plus générale $l(\mathrm{F}_{n+1},\mathrm{F}_n) = n$, et bien entendu $l(d\mathrm{F}_{n+1},d\mathrm{F}_n) = n$ pour tout entier naturel non nul d.

Lemme 1. *Si $d :=$ pgcd (a,b) (avec $a \geq b > 0$) et $l(a,b) = n$, alors $a \geq d\mathrm{F}_{n+1}$.*

Preuve : On montre les inégalités $a \geq d\mathrm{F}_{n+1}$ et $b \geq d\mathrm{F}_n$ par récurrence sur n. Si $n = 1$, b divise a et $d = b$, donc $a \geq d\mathrm{F}_2$ et $b \geq d\mathrm{F}_1$. Si la propriété est vraie au rang n, et si $n + 1$ divisions sont nécessaires pour calculer $d =$ pgcd (a,b), la première division s'écrit $a = bq + r$, puis la seconde division est celle de b par r. Le nombre de divisions nécessaires au calcul de pgcd (b,r) est n, et l'hypothèse récurrente implique $b \geq d\mathrm{F}_{n+1}$ et $r \geq d\mathrm{F}_n$. On obtient bien $a = bq + r \geq d\mathrm{F}_{n+1} + d\mathrm{F}_n = d\mathrm{F}_{n+2}$ et $b \geq d\mathrm{F}_{n+1}$.

Théorème 1. *Si $a > b > 0$, alors* $l(a,b) \leq \dfrac{3}{2}\log_2 a + 1$.

Preuve : Le Lemme permet d'écrire

$$a \geq d\,\mathrm{F}_{n+1} \geq \mathrm{F}_{n+1} \Rightarrow \log_2 a \geq \log_2 \mathrm{F}_{n+1}$$

et il suffit de vérifier l'inégalité $\log_2 \mathrm{F}_{n+1} \geq \frac{2}{3}(n-1)$ pour conclure. Si $\varphi = \frac{1+\sqrt{5}}{2}$ désigne le « nombre d'or », on sait que $\mathrm{F}_n = \frac{1}{\sqrt{5}}\left[\varphi^n - (1-\varphi)^n\right]$. Par conséquent

$$\mathrm{F}_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right] = \frac{\left(1+\sqrt{5}\right)^n}{2^n\sqrt{5}}\left[1-\left(\frac{1-\sqrt{5}}{1+\sqrt{5}}\right)^n\right]$$

donc

$$\log_2(\mathrm{F}_{n+1}) = (n+1)\log_2\left(1+\sqrt{5}\right) + \log_2\left[1-\left(\frac{1-\sqrt{5}}{1+\sqrt{5}}\right)^{n+1}\right] - (n+1) - \frac{\log_2 5}{2}.$$

Comme $\frac{1-\sqrt{5}}{1+\sqrt{5}} \cong -0{,}38$, on obtient :

$$\log_2\left[1-\left(\frac{1-\sqrt{5}}{1+\sqrt{5}}\right)^{n+1}\right] \geq \log_2\left[1-\left(\frac{1-\sqrt{5}}{1+\sqrt{5}}\right)^{2}\right]$$

et l'on aura prouvé l'inégalité $\log_2 \mathrm{F}_{n+1} \geq \frac{2}{3}(n-1)$ si l'on démontre que l'expression

$$\mathrm{A} := (n+1)\log_2\left(1+\sqrt{5}\right) + \log_2\left[1-\left(\frac{1-\sqrt{5}}{1+\sqrt{5}}\right)^{2}\right] - (n-1) - \frac{\log_2 5}{2} - \frac{2(n-1)}{3}$$

est positive pour tout $n \in \mathbf{N}^*$. On trouve $\mathrm{A} = \mathrm{U}n + \mathrm{V}$ avec

$$\mathrm{U} = \log_2\left(1+\sqrt{5}\right) - \frac{5}{3}$$

et

$$\mathrm{V} = \log_2\left(1+\sqrt{5}\right) + \log_2\left[1-\left(\frac{1-\sqrt{5}}{1+\sqrt{5}}\right)^{2}\right] - \frac{1}{3} - \frac{\log_2 5}{2}$$

et l'on vérifie que $\mathrm{U} \cong 0{,}028$ et $\mathrm{U} + \mathrm{V} \cong 2 \times 10^{-9}$ sont des réels strictement positifs.

4.3. Coût

Posons commodément $a = r_{-1}$ et $b = r_0$, puis notons C (k) le coût de la k-ième division $r_{k-2} = r_{k-1}q_k + r_k$ de l'algorithme d'Euclide **(AE)**. On a (Section 3.2) :

$$\mathrm{C}(k) = c_{\div}(\log_2 r_{k-2})(1 + \log_2 r_{k-2} - \log_2 r_{k-1}).$$

Le coût C_{AE} de l'algorithme **(AE)** sera donc successivement égal à :

$$c_{\div}[(\log_2 a)(1 + \log_2 a - \log_2 b) + (\log_2 b)(1 + \log_2 b - \log_2 r_1) \\ + \ldots + (\log_2 r_{n-2})(1 + \log_2 r_{n-2} - \log_2 r_{n-1})]\,,$$

$$c_{\div}\left[\sum_{k=1}^{n-2} \log_2 r_k + (\log_2 a)^2 - \sum_{k=0}^{n-2} (\log_2 r_k)(\log_2 r_{k-1} - \log_2 r_k) - \log_2 r_{n-2} \log_2 r_{n-1}\right].$$

Le terme $\log_2 r_{n-2} \log_2 r_{n-1}$ ainsi que tons les termes de la seconde somme du membre de droite sont positifs, donc

$$\mathrm{C}_{AE} \leq c_{\div}\left[(n-1)\log_2 a + (\log_2 a)^2\right].$$

Compte tenu du Théorème 1, on obtient :

$$\mathrm{C}_{AE} \leq \frac{5}{2} c_{\div}(\log_2 a)^2.$$

On a montré :

Théorème 2. *Dans le modèle à coûts bilinéaires, le coût de l'algorithme d'Euclide de calcul du pgcd de a et b (avec a ≥ b > 0) est majoré par* $\frac{5}{2} c_{\div}(\log_2 a)^2$.

L'algorithme est donc de complexité $\mathrm{O}\left((\log_2 a)^2\right)$.

5. Algorithme d'Euclide étendu

5.1. Principe

L'algorithme d'Euclide étendu propose non seulement d'obtenir le pgcd d de a et b, mais aussi de fournir les coefficients entiers u et v tels que $d = au + bv$. À chaque division de l'algorithme (**AE**), associons les entiers u_k et v_k – s'ils existent – tels que $r_k = au_k + bv_k$. On a $a = bq_1 + r_1$, donc $(u_1,v_1) = (1,-q_1)$. On a ensuite $b = r_1q_2 + r_2$, donc

$$r_2 = b - r_1q_2 = b - (a - bq_1)q_2 = a(-q_2) + b(1 + q_1q_2)$$

et $(u_2,v_2) = (-q_2,1 + q_1q_2)$. Si les écritures des restes $r_1, \ldots, r_{k-1}$ comme combinaisons linéaires à coefficients entiers des nombres a et b sont possibles, alors r_k s'exprime encore sous la forme :

$$r_k = r_{k-2} - r_{k-1}q_k = a(u_{k-2} - u_{k-1}q_k) + b(v_{k-2} - v_{k-1}q_k) = au_k + bv_k,$$

en posant

$$(u_k, v_k) = (u_{k-2} - u_{k-1}q_k, v_{k-2} - v_{k-1}q_k).$$

L'existence de décompositions de la forme $r_k = au_k + bv_k$ est donc assurée par récurrence, et $d = r_{n-1} = au_{n-1} + bv_{n-1}$.

La mise en œuvre de l'algorithme d'Euclide étendu consiste à reprendre les instructions de l'algorithme d'Euclide en y ajoutant les calculs de u_k et v_k à chacune des divisions. On obtient l'algorithme d'Euclide étendu symétrique dont voici une implémentation en Maple :

(AEES) : Algorithme d'Euclide étendu symétrique
Entrées : deux entiers relatifs quelconques adata et bdata.
Sorties : d, u, v où d = pgcd(adata,bdata) et d = au + bv.

```
>pgcdets:=proc(adata,bdata);
>a:=adata ; b:=bdata;
>if b=0 then a, 1, 0 else
>q:=iquo(a,b) ; r:=a–q*b;
>if r=0 then b, 0, 1 else
>u1:=1 ; v1:=–q;
>a:=b ; b:=r ; q:=iquo(a,b) ; r:=a-q*b ; u2:=q*u1 ; v2:=1–q*v1;
>while r<>0 do
>a:=b ; b:=r ; q:=iquo(a,b) ; r:=a-q*b;
>u:=u1–q*u2 ; v:=vl–q*v2 ; u1:=u2 ; v1:=v2; u2:=u ; v2:=v;
>od;
>b, u1, v1;
>fi ; fi;
>end;
```

En fait, ce premier algorithme est inutilement compliqué puisqu'il suffit de calculer d et l'entier u tel que $d = au + v$, pour obtenir v par division euclidienne exacte. On peut donc recopier l'algorithme précédent en « gommant » toutes les références à la suite (v_k) pour obtenir l'algorithme d'Euclide étendu dissymétrique suivant :

(AEED) : Algorithme d'Euclide étendu dissymétrique
Entrées : deux entiers relatifs quelconques adata et bdata.
Sorties : d, u, v où d = pgcd(adata,bdata) et d = au + bv.

```
>pgcdetd:=proc(adata,bdata);
>a:=adata ; b:=bdata;
>if b=0 then a, 1, 0 else
>q:=iquo(a,b) ; r:=a–q*b;
>if r=0 then b, 0, 1 else
>u1:=1;
>a:=b ; b:=r ; q:=iquo(a,b) ; r:=a-q*b ; u2:=q*u1;
>while r<>0 do
>a:=b ; b:=r ; q:=iquo(a,b) ; r:=a-q*b;
>u:=u1–q*u2 ; u1:=u2 ; u2:=u;
>od;
>b, u1, iquo(b–(adata*u1),bdata);
>fi ; fi;
>end;
```

Les deux algorithmes permettent d'inverser une classe $\bar{a}$ dans **Z**/n**Z**. En effet, la classe $\bar{a}$ est inversible dans **Z**/n**Z** si et seulement si a est premier avec n, et le calcul de u et v tels que $au + nv = 1$ donne $\bar{a}\bar{u} = \bar{1}$. Si le but est seulement de calculer l'inverse de a modulo n, on peut adapter l'algorithme **(AEED)** en y supprimant le calcul de iquo(b-(adata*u1),bdata). On obtient alors un algorithme légèrement plus court que nous appellerons **(AI)** (algorithme d'inversion modulo n).

5.2. Coûts

Lemme 2. *Avec les notations de la Section 5.1, la suite* (u_k) *est alternée, de valeurs absolues strictement croissantes et* $du_n = (-1)^n b$.

Preuve : On a $u_1 = 1$ et $u_2 = -q_2$. Tous les coefficients q_k intervenant dans l'algorithme sont ≥ 1 (sinon $q_k \leq 0$ entraîne $r_{k-2} = r_{k-1}q_k + r_k \leq r_k$ en contradiction avec la stricte décroissance de $(r_k)_{1\leq k\leq n}$), donc $|u_2| \geq |u_1|$. Soit H(k) la propriété « $u_1, \ldots, u_k$ est alternée et de valeurs absolues strictement croissantes ». On vient de vérifier que H(2) est vraie. Si H($k-1$) est vérifiée, alors

$$|u_k| = |u_{k-2} - u_{k-1}q_k| = |u_{k-2}| + |u_{k-1}|q_k \geq |u_{k-2}| + |u_{k-1}| > |u_{k-1}|,$$

et

$$u_k \times u_{k-1} = u_{k-2}u_{k-1} - u_{k-1}^2 q_k < 0,$$

de sorte que H(k) soit vraie. Par ailleurs

$$\begin{aligned} r_k u_{k+1} - r_{k+1}u_k &= r_k(u_{k-1} - u_k q_{k+1}) - r_{k+1}u_k \\ &= r_k u_{k-1} - (r_k q_{k+1} + r_{k+1})u_k = r_k u_{k-1} - r_{k-1}u_k , \end{aligned}$$

donc

$$\begin{aligned} r_k u_{k+1} - r_{k+1}u_k &= -(r_{k-1}u_k - r_k u_{k-1}) \\ &= \ldots = (-1)^{k-1}(r_{-1}u_0 - r_0 u_{-1}) = (-1)^{k-1} b , \end{aligned}$$

en posant $r_0 = b$ et $r_{-1} = a$. Il suffit de faire $k = n - 1$ dans cette dernière formule pour obtenir $du_n = (-1)^n b$.

Remarque : On obtiendrait de même $dv_n = (-1)^n a$.

Le coût des algorithmes étendus est égal à celui du calcul des n divisions euclidiennes augmenté du coût des calculs des entiers $u_k = u_{k-2} - u_{k-1}q_k$ et $v_k = v_{k-2} - v_{k-1}q_k$. Le coût du calcul de $u_k = u_{k-2} - u_{k-1}q_k$ est :

$$c_+ \sup(\log_2 |u_{k-2}|, \log_2 |u_{k-1}q_k|) + c_\times(\log_2 |u_{k-1}|)(\log_2 q_k)$$

majoré par $(c_+ + c_\times)(\log_2 |u_{k-1}|)(\log_2 q_k)$, ou encore par $(c_+ + c_\times)(\log_2 a)(\log_2 q_k)$ d'après le Lemme 2, et en supposant $a \geq b > 0$. Le coût $\mathrm{C}_{\mathrm{AEES}}$ de l'algorithme **(AEES)** sera donc majoré par :

$$\mathrm{C}_{\mathrm{AE}} + 2(c_+ + c_\times)(\log_2 a)\sum_{k=1}^{n} \log_2 q_k .$$

Comme $a \geq bq_1 \geq r_1 q_1 q_2 \geq \ldots \geq q_1 q_2 \ldots q_n$, on obtient :

$$\sum_{k=1}^{n} \log_2 q_k = \log_2\left(\prod_{k=1}^{n} q_k\right) \leq \log_2 q_k\,,$$

soit :

$$C_{\mathrm{AEES}} \leq C_{\mathrm{AE}} + 2(c_+ + c_\times)(\log_2 a)^2.$$

Compte tenu du Théorème 2 et du fait que l'algorithme **(AI)** de recherche de l'inverse modulo n ne nécessite que le calcul des u_k, on obtient :

Théorème 3. *On suppose $a \geq b > 0$. Les coûts C_{AEES} et C_{AI} des algorithmes* **(AEES)** *et* **(AI)** *vérifient :*

$$C_{\mathrm{AEES}} \leq \left(\frac{5}{2} c_\div + 2c_+ + 2c_\times\right)(\log_2 a)^2$$

et

$$C_{\mathrm{AI}} \leq \left(\frac{5}{2} c_\div + c_+ + c_\times\right)(\log_2 a)^2.$$

Le coût de l'algorithme **(AEED)** sera égal à C_{AI} augmenté du coût C du calcul de $v_{n-1} = \dfrac{d - au_{n-1}}{b}$. On a :

$$\begin{aligned} C \leq{} & c_\times(\log_2 a)(\log_2 |u_{n-1}|) + c_+ \sup(\log_2(d), \log_2(a|u_{n-1}|)) \\ & + c_\div(\log_2(|d - au_{n-1}|))(1 + \log_2(|d - au_{n-1}|) - \log_2 b). \end{aligned}$$

On a $|u_{n-1}| \leq b \leq a$ et $|v_{n-1}| \leq a$ d'après le Lemme 2 et la remarque qui le suit, et $d - au_{n-1} = bv_{n-1}$, donc :

$$\begin{aligned} C &\leq c_\times(\log_2 a)^2 + 2c_+(\log_2 a) + c_\div(\log_2(b|v_{n-1}|))(1 + \log_2(b|v_{n-1}|) - \log_2 b) \\ &\leq c_\times(\log_2 a)^2 + 2c_+(\log_2 a) + c_\div(2\log_2 a)(1 + \log_2 a) \\ &\leq (2c_+ + 2c_\div)\log_2 a + (c_\times + 2c_\div)(\log_2 a)^2. \end{aligned}$$

En écrivant $C_{\mathrm{AEED}} \leq C_{\mathrm{AI}} + C$, on obtient une majoration qui laisse à penser que l'algorithme **(AEED)** n'est pas meilleur que **(AEES)** dès que l'on veut à la fois u et v, mais cette affirmation dépend de la majoration que l'on a su donner de C. On retiendra seulement que les algorithmes d'Euclide **(AE)**, **(AEES)**, **(AI)** et **(AEED)** sont tous de complexité $O\left((\log_2 a)^2\right)$.

Théorème 4. *On suppose $a \geq b > 0$. Le coût C_{AEED} de l'algorithme* **(AEED)** *vérifie :*

$$C_{\mathrm{AEED}} \leq (2c_+ + 2c_\div)\log_2 a + \left(\frac{9}{2} c_\div + c_+ + 2c_\times\right)(\log_2 a)^2.$$

6. Compléments

6.1. Une alternative au Lemme 1

Le document d'accompagnement des programmes [2] du CNDP propose une activité complète sur la suite de Fibonacci (p. 18-21) au niveau de l'enseignement de spécialité de terminale S. Un encart précise le rôle de cette suite dans l'étude de l'algorithme d'Euclide, et se contente de démontrer l'inégalité $a \geq \mathrm{F}_{n+1}$ au lieu de l'inégalité $a \geq d\,\mathrm{F}_{n+1}$ du Lemme 1. Cette inégalité suffit à montrer le Théorème 1, et permet d'exposer une méthode sans doute plus « constructive » si on le désire, basée sur les deux écritures suivantes :

$$(1) \quad \begin{cases} \mathrm{F}_{n+1} = \mathrm{F}_n + \mathrm{F}_{n-1} \\ \mathrm{F}_n = \mathrm{F}_{n-1} + \mathrm{F}_{n-2} \\ \cdots\cdots \\ \mathrm{F}_3 = \mathrm{F}_2 + \mathrm{F}_1 \\ \mathrm{F}_2 = \mathrm{F}_1 \end{cases}$$

et

$$(2) \quad \begin{cases} a = bq_1 + r_1 \geq b + r_1 \\ b = r_1 q_2 + r_2 \geq r_1 + r_2 \\ r_1 = r_2 q_3 + r_3 \geq r_2 + r_3 \\ \cdots\cdots \\ r_{k-2} = r_{k-1} q_k + r_k \geq r_{k-1} + r_k \\ \cdots\cdots \\ r_{n-2} = r_{n-1} q_n \geq r_{n-1} \end{cases}$$

L'écriture (1) montre que l'algorithme d'Euclide appliqué au couple $(\mathrm{F}_{n+1},\mathrm{F}_n)$ aboutit en n étapes. Dans l'écriture (2) on observe un couple quelconque (a,b) tel que l'algorithme fonctionne en n étapes, et il est facile de déduire l'inégalité $a \geq \mathrm{F}_{n+1}$. En effet, montrons par récurrence que la propriété $r_{n-k-1} \geq \mathrm{F}_{k+1}$ est vraie pour tout k appartenant à $\{1, \ldots, n\}$. La propriété est triviale au rang $k = 1$, et si elle est vraie jusqu'au rang $k - 1$ (avec $k \leq n$),

$$r_{n-k-1} \geq r_{n-k} + r_{n-k+1} \geq \mathrm{F}_k + \mathrm{F}_{k-1} = \mathrm{F}_{k+1}.$$

On vient de prouver que F_{n+1} est le plus petit entier naturel a pour lequel l'algorithme fonctionne en n étapes.

6.2. Écriture d'un entier en base 2

Soit m un entier naturel non nul. L'algorithme des divisions successives de m par 2 fournit une preuve constructive de l'existence d'une décomposition de m en base 2, c'est-à-dire sous la forme :

$$m = a_k 2^k + a_{k-1} 2^{k-1} + \ldots + a_1 \times 2 + a_0$$

où $a_{k-1}, \ldots, a_1, a_0 \in \{0,1\}$ et $a_k = 1$. Notons $m = \overline{a_k a_{k-1} \ldots a_0}$ l'écriture de m en base 2. Par exemple, les divisions

$$\begin{array}{cccccccccccc}
43 & | & 2 & & & & & & & & & \\
1 & | & 21 & | & 2 & & & & & & & \\
 & & 1 & | & 10 & | & 2 & & & & & \\
 & & & & 0 & | & 5 & | & 2 & & & \\
 & & & & & & 1 & | & 2 & | & 2 & \\
 & & & & & & & & 0 & | & 1 &
\end{array}$$

fournissent une suite de restes qui, lue « à l'envers », permettent d'écrire $43 = \overline{101011}$. De façon générale, on écrit les divisions

$$\begin{cases}
m & = 2q_0 + r_0 & \\
q_0 & = 2q_1 + r_1 & \times 2 \\
q_1 & = 2q_2 + r_2 & \times 2^2 \\
\vdots & \vdots & \vdots \\
q_{k-1} & = 2q_k + r_k & \times 2^k
\end{cases}$$

où $r_i \in \{0, 1\}$ pour tout i. Puis on remarque qu'il existe forcément un moment où l'un des q_i s'annule. En effet, si tous les quotients q_i étaient strictement positifs, la suite des q_i serait strictement décroissante dans **N**, ce qui est absurde. Dans le tableau ci-dessus on peut donc supposer $q_k = 0$ et $q_{k-1} \neq 0$. Ii suffit de multiplier les lignes du tableau par les puissances de 2 indiquées et de sommer pour obtenir :

$$m = r_k 2^k + \ldots + r_2 2^2 + r_1 2 + r_0$$

avec $r_k, \ldots, r_0 \in \{0, 1\}$ et $r_k = 1$.

Avec MuPad, le programme peut s'écrire:

```
> m:=25 ; q:=m;
> while q>0 do r:= q mod 2 ; q:= q div 2 ; print(r) ; end_while;
```

et donne la suite des restes à lire « à l'envers ».

Le coût de la i-ème division $q_{i-1} = 2q_i + r_i$ est

$$\mathrm{C}(i) = c_{\div}(\log_2 q_{i-1})(1 + \log_2 q_{i-1} - \log_2 q_i)$$

et le coût $\mathrm{C} = \sum_{i=0}^{k-1} \mathrm{C}(i)$ des $k = [\log_2 m]$ divisions de l'algorithme complet sera majoré comme en 4.3 :

$$C \leq c_{\div}\left[k \log_2 m + (\log_2 m)^2\right] \leq 2c_{\div}(\log_2 m)^2.$$

Références

[1] L. Adleman, R.L. Rivest & A. Shamir, A method for Obtaining Digital Signatures and Public-Key Cryptosystems, Communications of the ACM, vol. **21**, Number 2, 1978, p. 120-126.

[2] Mathématiques, classes terminales de la série scientifique et de la série économique et sociale, Collection Lycée, série « Accompagnement des programmes », Ministère de la Jeunesse, de l'Éducation Nationale et de la Recherche, CNDP, juillet 2002. *Ce document a été rédigé par un groupe d'experts sur les programmes scolaires de mathématiques*.

[3] M. Demazure, Cours d'Algèbre, Primalité, Divisibilité, Codes, Éditions Cassini, 1997.

[4] D.-J. Mercier, Cryptographie Classique et Cryptographie Publique à Clé Révélée, Bulletin de l'APMEP **406**, p. 568-581, septembre 1996.

[5] J. Shallit, Origins of the analysis of the euclidean algorithm, Historia Mathematica **21**, p. 401-419, 1994.