COMPLEXITE D'UN ALGORITHME : UNE QUESTION CRUCIALE ET ABORDABLE

Gilles ALDON, Jérôme GERMONI
et Jean-Manuel MENY

Irem de Lyon

Introduction

Les derniers programmes de mathématiques (BO n°30 du 3 juillet 2009 et, plus récemment, JO n° 0199 du 28 août 2010) du lycée insistent sur « la place naturelle dans tous les champs des mathématiques » de l'algorithmique ; les objectifs annoncés sont de conduire les élèves, tout au long de leur scolarité au lycée, à décrire, interpréter et traduire des algorithmes. En particulier, les programmes évoquent leur mise en pratique « sur une calculatrice ou à l'aide d'un logiciel » (*ibid*. page 9). Un groupe de l'Irem de Lyon s'est constitué à la rentrée 2009 pour réfléchir à la mise en œuvre de ces objectifs du programme dans les classes de lycée, et cette réflexion a débouché sur l'organisation d'un stage de formation continue ; cet article reprend les idées directrices de ce stage dans lequel il ne s'agissait pas d'élaborer une introduction à l'algorithmique en classe de seconde mais plutôt, à partir d'exemples, de présenter des notions de l'algorithmique qui apparaissaient comme fondamentales et indispensables à la culture des professeurs de mathématiques.

Le choix de travailler sur les algorithmes et de les mettre en œuvre sur Xcas a été fait pour suivre les recommandations des programmes de mathématiques, tout en insistant sur la distinction entre algorithmique et programmation ; dans cet esprit, la devise du site Computer Science Unplugged (http://csunplugged.org/) rappelle que le travail sur l'algo-

rithmique ne nécessite pas le travail sur machine : « Teach computer science without a computer! » Si récente qu'elle soit, l'histoire de l'enseignement de l'algorithmique met en évidence différents paradigmes d'enseignement. Ils ont été bien relevés et décrits par Chi Thanh Nguyen dans sa thèse (Chi Than Nguyen, 2009) et montrent bien les évolutions, dans l'enseignement des relations entre l'algorithmique et la programmation.

Dans les années cinquante, Kuntzman utilise une machine « évoquée » mais réelle (EDSAC) : « Dans cet enseignement dit de programmation, l'auteur établit un lien étroit entre la programmation, l'architecture de la machine, son fonctionnement et l'écriture des programmes à une machine donnée » (Nguyen, 2005, page 44). Plus tard, Horowitz et Sahni (1978) choisissent de créer un langage universel de description d'algorithmes : « Plus le langage est suffisamment proche de plusieurs langages existants déjà mentionnés (Algol, Fortran, Lips, Pascal et PL/I, etc.), plus une traduction à la main est relativement facile à accomplir.

Ceci nous encourage à développer un langage simple qui est taillé pour la description des algorithmes qui vont être discutés » (Horowitz et Sahni, 1978, cité par Nguyen). Enfin, chez Knuth (*The Art of Computer Programming*, 1968), la machine évoquée est une machine idéale, représentation de l'ensemble des machines (MIX et son langage MIXA). Nous avons choisi, dans cet article comme dans le stage, d'écrire les algorithmes en « pseudo-langage » pour permettre une traduction aisée en de vrais programmes et de proposer une implémentation sur Xcas par des fichiers disponibles sur le site de la revue *Repères-IREM*. Dans cet article, nous souhaitons discuter de trois questions fondamentales qui se posent lorsqu'on construit un algorithme :

- l'algorithme donne-t-il un résultat ? c'est la question de la terminaison ;
- l'algorithme donne-t-il le résultat souhaité ? c'est la question de la validité ou de la correction ;
- l'algorithme donne-t-il ce résultat dans un temps raisonnable ? c'est la question de la complexité.

Ces questions sont discutées à partir des exemples utilisés en formation. Nous laissons au lecteur le soin de les adapter pour la classe.

Terminaison : la notion de convergent

Quand est-on sûr qu'un algorithme se termine ? C'est évidemment le cas lorsque l'on fixe *a priori* le nombre d'étapes, c'est-à-dire lorsque la longueur du processus est connue à l'avance ; un exemple de cette situation est donné par les algorithmes reposant sur une boucle finie :

```
pour i allant de 1 jusque 1000 faire
    ...
```

Dans l'algorithme habituel d'addition de deux entiers, par exemple, la terminaison est claire car le nombre d'opérations sur les chiffres est au plus le nombre de chiffres du plus grand terme (plus un s'il y a une retenue à ajouter).

Dans la plupart des problèmes, cependant, le nombre d'étapes n'est pas défini dans l'algorithme. Typiquement, pour sortir d'une boucle « tant que », il faut s'assurer que la condition qui suit l'instruction « tant que » est satisfaite après un certain nombre de passages dans la boucle. Ce nombre peut être très grand et, dans la plupart des cas, on ne le connaît pas à l'avance.

Par exemple (canonique), dans l'algorithme d'Euclide pour le calcul du pgcd de deux

entiers naturels, chaque passage dans la boucle fait diminuer strictement le deuxième des deux nombres, ce qui force *in fine* la fin de l'algorithme. On trouvera plus de détails ci-dessous. Cette situation est typique : il est fréquent que l'on prouve la terminaison d'un algorithme en exhibant une quantité entière qui diminue strictement à chaque étape. Une telle quantité est appelée un *convergent*. Un convergent n'est pas nécessairement une donnée du problème, voir l'exemple des fractions égyptiennes ci-dessous.

Il peut arriver aussi que le convergent ne soit pas un entier mais, comme dans les exemples 5 et 6 ci-dessous, un couple d'entiers naturels dans l'ensemble $\mathbf{N} \times \mathbf{N}$ que l'on ordonne par l'ordre lexicographique : par définition, étant donné deux couples (n,p) et (n',p'), on dit que

$(n,p) < (n',p')$ si $[n < n'$ ou $(n = n'$ et $p < p')]$.

L'ordre lexicographique partage avec l'ordre naturel sur $\mathbf{N}$ la propriété d'être un *bon ordre*, ce qui signifie que toute suite décroissante au sens large contient deux termes égaux. De façon équivalente, il n'existe pas de suite infinie strictement décroissante ou encore, toute partie non vide possède un plus petit élément. Ainsi, si une quantité – un convergent – diminue strictement à chaque passage dans la boucle, on crée une suite strictement décroissante, qui par la propriété de bon ordre est nécessairement finie.

Correction : la notion d'invariant de boucle

Toutes les preuves de correction que nous allons faire reposent sur la notion d'*invariant de boucle*. Un invariant de boucle est une assertion qui est vraie avant l'entrée dans la boucle et reste vraie après chaque passage dans la boucle et, par conséquent, en sortie de boucle. Naturellement, cette assertion porte sur les variables de l'algorithme, leur valeur initiale ou leur valeur « courante ». La démarche d'une preuve de correction par l'utilisation d'un invariant de boucle est très semblable à une preuve par récurrence, où la condition en entrée de boucle joue le rôle de l'initialisation et la préservation de la validité au passage au cours d'un passage dans la boucle joue celui de l'hérédité.

Remarquons que la preuve porte sur l'algorithme et pas sur un programme écrit dans un langage précis sur une machine précise, ce qui est un autre problème. Il peut ainsi arriver que l'algorithme soit prouvé et que le programme bugue obstinément…

Des exemples

Exemple 1 : division euclidienne « par la droite »

Observons la méthode suivante pour faire la division de 999 999 par 7.

A chaque étape, on cherche le plus petit multiple[1] de 7 qui a le même chiffre des unités que le dividende courant. Par exemple, le multiple de 7 qui a 9 pour chiffre des unités, c'est $7 \times 7 = 49$. On garde 7 dans le quotient, on retranche 49 de 999 999, reste 999 950 dont on efface le zéro final.

En 99 995, combien de fois 7 ? Eh bien, 5 fois puisque $5 \times 7 = 35$; on retranche 35, reste 9996. En 9 996, combien de fois 7 ?

Eh bien, 8 fois comme chacun sait ; on retranche 56, reste 994. En 994, combien de fois 7 ? Eh bien, 2 fois, on retranche 14, reste 98 ; en 98 sont 4 fois 7, on retranche 28, reste 7. En

1 Comme 7 et 10 sont premiers entre eux, il existe bien un tel multiple ; de plus, on peut se contenter de regarder les produits par un nombre à un chiffre.

7 va 1 fois 7, reste 0. Au bilan, on a le bon quotient : 999 999 = 7 × 142 857 (qu'on a découvert de droite à gauche).

Etonnant, non ? Voici une présentation de cette opération plus concise en potence.

999 999	7
49	142 857
999 950	
35	
999 60	
5 6	
994 0	
14	
980	
28	
70	
7	
0	

Algorithme division_droite
Entrée : a entier, b entier non nul et non divisible par 2 ou 5
Sortie : le quotient entier de a par b
Corps :
  initialiser trois variables[2] r := a, q := 0, k := 0 ;
  tant que r ≠ 0 **faire**
    Trouver d dans {1,2,…,9} tel que b * d ait le « même chiffre des unités » que r
    r := (r – b * d) / 10 *// ici, l'étoile * désigne le produit*
    q := d * 10^k + q
    k := k + 1
  renvoyer q

L'algorithme a bien un sens. En effet, d'après les critères de divisibilité par 2 et 5 et les hypothèses, le chiffre des unités de b est premier avec 10, donc la multiplication par b induit une bijection de $\mathbf{Z}/10\mathbf{Z}$ dans lui-même.

De façon équivalente, plus élémentaire mais plus longue, le chiffre des unités de b est 1, 3, 7 ou 9 et tous les chiffres de 1 à 9 apparaissent une fois et une seule comme chiffre des unités dans chacune des tables de multiplication par 1, 3, 7 et 9. Ceci garantit l'existence d'un unique chiffre c à chaque étape.

Malheureusement, pour une division avec reste, l'algorithme ne donne pas la réponse espérée, c'est-à-dire le quotient et le reste de la division euclidienne, même si toutes les égalités que l'on obtient sont justes. Divisons par exemple 64 par 7 avec cette méthode : en 4 vont 2 fois 7, on retranche 14, reste 5 ; en 5 vont 5 fois 7, on retranche 35, reste –30…

Pas très satisfaisant, mais on peut continuer : en –3 = 7 va 1 fois 7 ; on retranche 7, reste –1 ; en –1 = 9 vont 7 fois 7, on retranche 49, reste –5 ; etc. Chaque étape de calcul correspond à une égalité vraie ; dans l'ordre, on a exprimé :

$$64 = 7 \times 2 + 50 = 7 \times 52 - 300 =$$
$$= 7 \times 152 - 1000 = 7 \times 7152 - 50000\ldots$$

Comme dans l'algorithme de division euclidienne habituel, on obtient un reste « de plus en plus petit » au sens où il est « de plus en plus divisible par 10 ».

Quand il « ne tombe pas juste », l'algorithme va se poursuivre indéfiniment. En tout état de cause, l'algorithme est valide et se termine lorsque le reste de la division euclidienne est nul (exercice : le démontrer !) mais il ne se termine pas en général. Cet algorithme, appelé *division aux puissances croissantes*, est plus familier dans le contexte des polynômes.

2 Ici, r sera le reste/le dividende ; q le quotient ; d le nouveau chiffre de q ; k le nombre de chiffres de q.

Exemple 2 : pgcd de deux entiers par l'algorithme d'Euclide

Présentation : Voici une première version de l'algorithme d'Euclide.

Algorithme pgcd
Entrée : a entier, b entier
Sortie : le pgcd de a et b, entier
Corps :
  initialiser deux variables annexes[3] u := a, v := b
  tant que v est différent de zéro, **faire** :
    z := v
    q := E(u/v) // *ici,* E *représente la partie entière*
    v := u – q * v // *ici,* * *représente le produit des entiers*
    u := z
  fin du tant que
  renvoyer u.

Terminaison : On peut justifier la terminaison de cet algorithme par le fait que la valeur de la variable v diminue strictement à chaque passage dans la boucle : en effet, on la remplace par le reste d'une division euclidienne par v. On crée ainsi une suite strictement décroissante d'entiers qui est nécessairement finie. Ainsi, la variable v est un convergent de l'algorithme d'Euclide.

Correction : Pour deux entiers u et v, notons D(u,v) l'ensemble des diviseurs communs à u et v. Dans l'algorithme pgcd, l'assertion suivante est un invariant de boucle : D(a,b) = D(u,v).

En effet, à l'initialisation, les variables a et u d'une part, b et v d'autre part, coïncident. L'effet d'un passage dans la boucle consiste à remplacer le couple (u,v) par le couple (v,u – qv). Or si un entier divise u et v, il divise aussi v et u – qv ; réciproquement, s'il divise v et u – qv, il divise u – qv + qv = u et v (et ce, quelle que soit la valeur de q). Ainsi, à la sortie de la boucle « tant que », les entrées a et b ont les mêmes diviseurs communs que les variables u et v. Comme on est sorti de la boucle, c'est que la valeur de la variable v est nulle ; il en résulte qu'à ce moment, on a : D(a,b) = D(u,0). L'ensemble D(u,0) possède un plus grand élément, u, c'est bien ce que renvoie l'algorithme. Non seulement la correction de l'algorithme d'Euclide est démontrée, mais elle constitue une preuve – constructive sil en est – de l'existence du pgcd de deux entiers.

Cet exemple, connu des élèves de lycée ou, du moins, accessible, met bien en évidence les ressorts permettant de démontrer la terminaison d'un algorithme, ressorts que l'on peut alors mettre en action dans d'autres cas.

Exemple 3 : recherche gloutonne de fractions égyptiennes

Présentation : Le problème est le suivant : exprimer un rationnel strictement compris entre 0 et 1 comme somme d'inverses d'entiers naturels tous distincts[4] ; en d'autres termes, étant donné deux entiers a et b avec $0 < a < b$, trouver une suite finie strictement croissante d'entiers $r_1, \ldots, r_s$ telle que $a/b = 1/r_1 + \ldots + 1/r_s$.

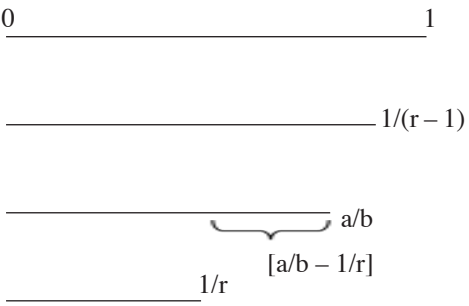
On pourra à ce propos consulter le cédérom EXPRIME (2010). Voici une méthode possible – un pré-algorithme, si on veut :

3 La création de deux variables annexes u et v que l'on manipule à la place de a et b n'est pas nécessaire. Au contraire, elle contrarie l'exécution puisqu'elle demande du temps et de l'espace pour copier les valeurs. Cependant, elle facilitera l'expression de la preuve de terminaison et de l'invariant de boucle.

4 Sans cette dernière contrainte, le problème est rendu trivial par l'écriture a/b = 1/b + … + 1/b.

Pré-algorithme egypte
Entrée : a entier, b entier tels que 0 < a < b
Sortie : une liste $[r_1,\dots,r_s]$ telle que $a/b = 1/r_1 + \dots + 1/r_s$
Méthode :
Trouver le plus petit entier r tel que 1/r < a/b
Remplacer a/b par a/b – 1/r
Recommencer jusqu'à trouver 0

Cette méthode peut être visualisée de la manière suivante :



Le premier travail consiste à traduire cette idée en pseudo-code. Plutôt que travailler avec une fraction a/b, on travaille avec les deux entiers a et b ; on utilise d'autres (noms de) variables u et v pour éviter que l'appel à l'algorithme modifie les variables que l'on est en train d'utiliser, ce qui risque de produire des effets de bord indésirables ; initialement, u vaut a et v vaut b. Enfin, on introduit une liste L qui va contenir les entiers r_i.

L'entier r est désormais caractérisé comme étant « le plus petit entier r tel que 1/r < u/v ». On suppose que u n'est pas nul, sinon il n'y a pas de question. La condition 1/r < u/v équivaut alors à r > v/u, pour autant que u ne soit pas nul. Si v/u est un entier, on doit prendre r = v/u ; si v/u n'est pas un entier, on doit prendre pour r la partie entière de v/u plus 1. La fonction mathématique que l'on vient de décrire peut être appelée « partie entière supérieure », *ceiling* en anglais : on va pour cette raison la noter ceil. Si E désigne la partie entière usuelle, on a pour tout réel A la relation : ceil(A) = – E(–A). On suppose qu'on dispose d'une implémentation de cette fonction (c'est le cas sur Xcas), on la notera toujours ceil.

Pré-algorithme egypte, version 2
Entrée : a entier, b entier tels que 0 < a < b
Sortie : une liste $[r_1,\dots,r_s]$ telle que $a/b = 1/r_1+\dots+1/r_s$
Méthode :
u := a ; v := b
tant que u ≠ 0 *// hypothèse de nécessité, dirait-on au bridge !*
poser r = ceil(v/u)
remplacer u/v par u/v – 1/r = (u*r – v)/(v*r)

En prenant en compte la nécessité de stocker les valeurs de r dans une liste, on obtient l'algorithme suivant en pseudo-code.

Algorithme egypte
Entrée : a entier, b entier tels que 0 < a < b
Sortie : une liste $[r_1,\dots,r_s]$ telle que $a/b = 1/r_1+\dots+1/r_s$
Corps :
initialiser u := a, v := b, L := [] *// ici, la notation [] désigne la liste vide*
tant que u ≠ 0 **faire**
r := ceil(v/u)
u := u * r – v *// ici, l'étoile * représente la multiplication…*
v := v * r
ajouter r à la liste L
fin du tant que
renvoyer L

On peut à présent programmer cet algorithme sur la machine de son choix. La traduction du pseudo-code ci-dessus à Xcas, par exemple, est quasiment immédiate. On peut faire quelques essais, ils fonctionnent – miraculeusement ? La question de la terminaison de cet algorithme et la preuve de sa validité ne vont pas de soi.

Terminaison : Montrons que la variable u est un convergent de l'algorithme. Il s'agit de montrer que sa valeur diminue strictement à chaque passage dans la boucle. Pour cela, on utilise la double inégalité qui caractérise la partie entière supérieure :

$$r - 1 < v/u \leq r,$$

d'où l'on tire en chassant le dénominateur :

$$0 \leq ur - v < u.$$

Cette double inégalité exprime que l'entier ur – v est positif ou nul et qu'il est strictement plus petit que u : comme c'est la nouvelle valeur de la variable u après passage dans la boucle, ces inégalités prouvent que u est bien un convergent. Notons au passage que la partie entière supérieure d'un réel positif ou nul est un entier strictement positif, si bien que la liste L ne contiendra que de tels entiers.

Correction : L'assertion suivante est un invariant de boucle :

$$\frac{a}{b} = \frac{u}{v} + \sum_{s \in L} \frac{1}{s}$$

La somme porte sur l'ensemble des éléments de la liste L. Par convention naturelle [5], la somme sur la liste vide est zéro. Avant d'entrer dans la boucle, le couple de variable (u,v) prend les mêmes valeurs que (a,b) et la liste L est vide donc l'assertion est vraie. Supposons qu'elle soit vraie au début d'un passage dans la boucle. Au cours de la boucle :

— la variable u est remplacée par u' = ru – v ;
— la variable v est remplacée par v' = rv ;
— la liste L est remplacée par la liste M obtenue en lui adjoignant l'élément r.

On a donc :

$$\frac{u'}{v'} + \sum_{s \in M} \frac{1}{s} = \frac{ru - v}{rv} + \frac{1}{r} + \sum_{s \in L} \frac{1}{s} =$$

$$= \frac{u}{v} - \frac{1}{r} + \frac{1}{r} + \sum_{s \in L} \frac{1}{s} = \frac{u}{v} + \sum_{s \in L} \frac{1}{s} = \frac{a}{b}.$$

Ainsi, l'assertion reste vraie après le passage dans la boucle. Une fois qu'on est sorti de la boucle, ce qui arrive nécessairement comme on l'a vérifié ci-dessus, l'assertion qui constitue l'invariant de boucle reste vraie et, de plus, la variable u est nulle. Il en résulte que a/b est bien la somme des inverses des éléments de la liste finale L.

Pour prouver la validité de l'algorithme, il ne reste plus qu'à vérifier que la suite des valeurs de r est strictement croissante. Si on n'en est pas à la dernière étape, c'est-à-dire si on a : r > v/u ou, ce qui revient au même, si u' n'est pas nul, la valeur suivante de la variable r sera la partie entière supérieure r' de v'/u'. Or on a :

$$\frac{v'}{u'} - r = \frac{rv}{ru - v} - r =$$

$$\frac{2v - ru}{ru - v} r = \frac{2\frac{v}{u} - r}{r - \frac{v}{u}} r.$$

Comme on sait que : r < v/u+1, il vient :

$$2\frac{v}{u} - r > 2\frac{v}{u} - \frac{v}{u} - 1 = \frac{v}{u} - 1 > 0.$$

5 Elle est établie de sorte que la somme sur la réunion de deux ensembles disjoints soit la somme des sommes : pour cela, il est nécessaire que la somme sur l'ensemble vide soit zéro.

On en déduit que v'/u' > r, puis que :

$$r' - r \geq \frac{v'}{u'} - r > 0.$$

Ainsi, la valeur de r décroît strictement à chaque passage, d'où la correction de l'algorithme. Remarquons au passage que rien ne prouve que la décomposition donnée par cet algorithme soit la seule décomposition possible de a/b comme somme d'inverses d'entiers distincts. En général, il existe d'ailleurs plusieurs telles décompositions. Un autre problème (difficile) serait de déterminer, pour un rationnel donné, le nombre de ses décompositions en n fractions égyptiennes.

Dans cet exemple, la suite décroissante d'entiers est un petit peu plus cachée que dans l'algorithme d'Euclide mais le travail mathématique sur les entiers est encore accessible à des élèves de lycée. Notons que l'algorithme et les preuves de terminaison et de correction démontrent un résultat purement mathématique, à savoir que tout nombre rationnel inférieur à un peut se décomposer comme somme de fractions égyptiennes distinctes.

Bien sûr, les questions de terminaison peuvent être beaucoup plus complexes. En voici quelques exemples.

Exemple 4 : algorithme de Syracuse

Présentation : L'algorithme suivant est bien connu.

```
Algorithme Syracuse
Entrée : n entier naturel non nul
Sortie : rien (d'autre que la gloire d'être sorti…)
Corps :
  tant que n ≠ 1 faire
    si n est pair faire n := n/2
    sinon faire n := 3 * n + 1
                  // ici, l'étoile * représente
                  la multiplication…
    fin du si
  fin du tant que
```

Terminaison : Devenez célèbre : prouvez que l'algorithme se termine pour toute valeur de la variable d'entrée n.

Exemple 5 : pseudo-algorithme aléatoire de Gilles Dowek

Présentation : L'algorithme suivant nous a été présenté par Gilles Dowek à l'occasion d'une conférence[6]. Il repose sur une hypothèse farfelue, à savoir qu'on dispose d'un générateur de nombres entiers aléatoires non bornés. On notera alea() l'appel à cette fonction, elle renvoie un nombre entier au hasard. (On ne s'intéresse pas à la loi de probabilités obtenue pour cet objet qui n'a de toute façon pas d'existence informatique…)

```
Algorithme aleatoire
Entrée : n, p entiers naturels
Sortie : chaîne de caractère
Corps :
  tant que n ≠ 0 et p ≠ 0 faire
    si p > 0 faire
        p := p – 1
    sinon faire
        si n > 0 faire
                n := n – 1
                p := alea()
        fin du si
    fin du si
  fin du tant que
  renvoyer « Terminé ! »
```

6 Voir sur son site http://www.lix.polytechnique.fr/~dowek/Vulg/ordinaux.html.

Discussion du caractère irréaliste de cet algorithme : Cet algorithme n'a qu'un intérêt théorique puisque la taille des nombres qu'on peut représenter dans un ordinateur est finie. Si le générateur aléatoire est borné, c'est-à-dire qu'il renvoie des entiers compris entre 1 et un certain entier M, il est clair que l'algorithme se termine toujours : en effet, il faut au plus M passages dans la boucle pour faire diminuer la valeur de n de 1 donc il faut au plus Mn passages dans la boucle pour passer de (n,p) à (0,0).

L'implémentation de cet algorithme n'est pas une très bonne idée pour autant, outre qu'il tourne sans rien calculer… En effet, on peut très facilement fabriquer un générateur aléatoire qui produit des nombres entre 1 et, disons, 10^{18} : il suffit de tirer au hasard dix-huit chiffres entre 0 et 9. La valeur moyenne de p au premier tirage aléatoire, c'est donc $10^{18}/2$: à raison de 10^9 opérations « p := p–1 » par seconde, ce qui représente une fréquence de l'ordre du GHz , on constate qu'il faut environ $5x10^8$ secondes, soit 16 années environ, pour arriver à 0. L'ordre de grandeur n'est pas du tout raisonnable.

Terminaison : L'algorithme se termine pourtant. On peut en effet le paraphraser de la façon suivante :

Algorithme paraphrase_aleatoire
Entrée : n, p entiers naturels
Sortie : chaîne de caractère
Corps :
  attendre le temps prescrit par p
  pour k **de** 1 **jusque** n **faire**
    attendre un temps aléatoire
  fin du pour
  renvoyer « Terminé ! »

Interprétation : Ordonnons $\mathbf{N} \times \mathbf{N}$ par l'ordre lexicographique. On a : $(n,p-1) < (n,p)$ et $(n-1,\text{alea}()) < (n,0)$: ainsi, la suite des valeurs prises par (n,p) décroît strictement tant que la valeur (0,0) n'a pas été atteinte. Inversement, soit une suite décroissante de couples d'entiers (n_k,p_k) (k entier naturel), telle qu'on ait de plus : $(n_{k+1},p_{k+1}) < (n_k,p_k)$ pour tout k, sauf si $(n_k,p_k) = (0,0)$. On peut considérer que c'est la suite des valeurs prises par les variables (n,p) lors d'une exécution de l'algorithme à partir de (n_0,p_0) (à deux différences près : l'algorithme intercale des valeurs intermédiaires et il faut remplacer alea() par des valeurs prescrites par la suite initiale). Comme l'algorithme se termine, la suite atteint la valeur (0,0). Autrement dit, la terminaison de l'algorithme exprime que l'ordre lexicographique sur $\mathbf{N} \times \mathbf{N}$ est un bon ordre.

Exemple 6 : « petites suites de Goodstein », encore d'après Gilles Dowek

Présentation : L'algorithme suivant est proposé par Gilles Dowek [7]. Il constitue une version simplifiée des suites de Goodstein que l'on présentera plus bas. Fixons un entier naturel n, par exemple 5. On va construire une suite d'entiers de la façon suivante.

Si, à un certain point, n est nul, on renvoie 0 et l'algorithme s'arrête. Sinon, on écrit la suite de chiffres de n en base 2 ; dans l'exemple, c'est 101. On interprète cette suite de chiffres comme un nombre écrit en base 3 ; dans l'exemple : $101 = 3^2 + 0\times3 + 1$, c'est dix. On retranche 1. On écrit ce nombre en base 3, on l'interprète en base 4 ; dans l'exemple, neuf s'écrit 100 en base 3, ce qui donne 100 en base 4 soit 4^2. On retranche 1. On écrit en base 4, on interprète en base 5, on retranche 1.

Et ainsi de suite. Formalisons un peu.

7 Voir par exemple ses notes de conférences : http://www.lix.polytechnique.fr/~dowek/Vulg/ordinaux.html

Soit b un entier supérieur ou égal à 2. Pour la $(b-1)^e$ étape, on doit faire l'opération de changement de base de b à b + 1 : on commence par écrire l'expression de n en base b de la façon suivante :

$$n = \sum_{k=0}^{r} a_k b^k$$

où les a_k sont les chiffres de n en base b, c'est-à-dire des entiers compris entre 0 et b – 1, et on remplace n par :

$$f_b(n) = \sum_{k=0}^{r} a_k (b+1)^k$$

L'opération de retrancher 1 est représentée par la fonction g qui à tout entier n associe n – 1. L'algorithme consiste à appliquer successivement f_2, g, f_3, g, f_4, g, f_5, g, f_6, g, etc., jusqu'à ce que l'ordinateur explose ou renvoie la valeur 0.

Les fonctions f_b ont tendance à « tirer vers le haut » à grands pas (bottes de sept lieues) alors que g est un pas de fourmi « vers le bas ». Qui gagne ? Il est assez amusant de programmer cet algorithme et d'observer le comportement pour différentes valeurs de n. Pour n < 6, la réponse est immédiate. Pour n > 8, l'ordinateur semble tourner sans fin. En particulier, il est instructif de représenter les points $(b-1, n_b)$, où n_b est la valeur de la variable n à la $(b-1)^e$ étape. Les fichiers Xcas correspondants sont disponibles sur le site de *Repères-IREM*.[8]

Terminaison : Le point-clé de la preuve de terminaison consiste à ne pas céder à la tentation de convertir les nombres produits par l'algorithme en base 10. On a l'impression d'être en présence de nombres de plus en plus grands mais c'est un leurre : ils sont de plus en plus petits… pour l'ordre lexicographique !

Travaillons en effet sur la suite des chiffres de la variable n dans la base courante. Notons d le nombre de chiffres de la donnée initiale.

Observons alors l'effet des applications f_b et g : la première ne change pas les chiffres ; la deuxième transforme une suite de chiffres en une suite qui est strictement plus petite pour l'ordre lexicographique : cela résulte de la propriété bien connue qui permet de comparer deux entiers, ici n et n–1, lorsque l'on connaît leurs représentations dans une base donnée. Mais quitte à ajouter des zéros à gauche si nécessaire, on peut toujours considérer que la suite des chiffres est, à tout moment, un élément de $\mathbf{N}^d$.

Observons plus précisément l'algorithme de soustraction en base b : si on ôte 1 à n, un nombre non nul ayant d chiffres en base b, de deux choses l'une :

— soit le chiffre des unités de n, noté u, est supérieur ou égal à 1, auquel cas n – 1 s'écrit de la même façon que n en remplaçant simplement u par u – 1 ;

— soit l'écriture en base b de n se termine par k zéros (k entier strictement positif), notons-la n = [ab…ef0…0], avec f non nul, auquel cas on a : n – 1 = [ab…e(f – 1)(b – 1)…(b – 1)] : le dernier chiffre non nul est diminué de 1 alors que les k chiffres égaux à 0 sont remplacés par (b – 1). Si on ne prête pas attention à l'étape à laquelle on se trouve, c'est-à-dire si on ne garde pas trace de b, il revient presque au même d'écrire alea() à la place de (b – 1)…

En d'autres termes, l'algorithme de l'exemple 5 était, *mutatis mutandis*, l'algorithme de soustraction de 1 à un nombre à deux chiffres dans une base variable… Désormais, on voit bien comment conclure : la suite des chiffres de la variable n est un convergent de l'algorithme à valeurs dans $\mathbf{N}^d$, où d est le

8 Voir en ligne : http://www.univ-irem.fr/spip.php?article=71&id_numero=85.

nombre de chiffres de la donnée initiale et où l'ensemble $\mathbf{N}^d$ est muni de l'ordre lexicographique. On croira ou on démontrera sans peine que cet ordre est un bon ordre.

Voici une version plus formelle de l'argument. Soit n_0 la donnée initiale et d le nombre de chiffres de n_0 en base 2 et, plus précisément, $[a_{d-1}a_{d-2}\ldots a_1a_0]$ l'écriture de n_0 en base deux. Pour tout entier n inférieur ou égal à n_0, on note $C_b(n)$ la suite des chiffres de n en base b, c'est une d-liste d'entiers, un élément de $\mathbf{N}^d$. L'observation précédente revient à dire que l'on a :

— d'une part, $C_{b+1}(f_b(n)) = C_b(n)$;

— d'autre part, $C_b(g(n)) < C_b(n)$ si g(n) n'est pas nul.

Comme $\mathbf{N}^d$ est muni d'un bon ordre, la suite $(C_b(n))$, indexée par b, atteint la valeur (0,…,0).

Exemple 7 : suites de Goodstein, toujours d'après Gilles Dowek

Nous ne faisons qu'esquisser la description de ces suites, introduites dès les années quarante [9]. L'idée est analogue à l'exemple précédent mais plus complexe. Pour commencer, il faut imaginer ce que peut être l'écriture héréditaire d'un nombre n en base b : on commence par écrire n en base b de la façon habituelle : $n = \sum_{k=0}^{r} a_k b^k$, où les a_k sont compris entre 0 et $b - 1$; puis, on écrit chaque exposant k supérieur ou égal à b en base b :

$k = \sum_{j\geq 0} a_j^{(1,k)} b^j$, où les $a_j^{(1,k)}$ sont compris entre 0 et $b - 1$; on recommence avec tous les exposants j qui seraient supérieurs ou égaux à b et ainsi de suite. Par exemple, l'écriture récursive de 517 en base 2 s'obtient en écrivant d'abord : $517 = 2^9 + 2^2 + 2^0$.

L'écriture héréditaire de 9 est :

$$9 = 2^3 + 2^0 = 2^{2+1} + 2^0,$$

qu'il faudrait remplacer dans l'exposant de 517. A la $(b-1)^e$ étape de l'algorithme, on enchaîne deux opérations. L'opération « bottes de sept lieues » consiste à remplacer, dans l'écriture héréditaire en base b, toutes les occurrences de b par (b + 1). Partant de 517, à la première étape, on transforme donc l'exposant 9 en

$$3^{3+1} + 3^0 = 81 + 1 = 82,$$

et donc 517 est transformé en :

$3^{82} + 3^3 + 3^0 = 1\,330\,279\,464\,729\,113\,309\,844\,748\,891\,857\,449\,678\,437$.

L'opération « pas de fourmi » consiste, de nouveau, à retrancher 1. N'est-ce pas dérisoire ?

Nous n'avons pas fait l'exercice consistant à programmer la notation héréditaire. Cela ne sert de toute façon pas à grand-chose car le comportement des suites de Goodstein, qui sont les suites obtenues en itérant l'algorithme pour différentes valeurs initiales, commencent par exploser violemment, ce que suggère l'exemple ci-dessus. On peut pourtant démontrer, comme Goodstein en 1944, que l'algorithme se termine toujours à 0. C'est, cependant, invérifiable sur machine : par exemple, si la donnée initiale est 2, il faut $3.2^{402\,653\,211} - 1$ étapes pour aboutir. La démonstration [10] de terminaison de l'algorithme consiste à nouveau à

9 R. Goodstein, « On the restricted ordinal theorem », dans *Journal of Symbolic Logic*, 9 (1944), 33-41.

10 On pourra par exemple consulter http://fr.wikipedia.org/wiki/Th%C3%A9or%C3%A8me_de_Goodstein.

exhiber un convergent : cette fois-ci, il vit dans un ensemble d'*ordinaux* appelé ε_0 , bien plus gros que $\mathbf{N}^d$ mais qui possède quand même la propriété de bon ordre.

La preuve est plus compliquée que celle de l'exemple 5 mais reste relativement accessible (aux professeurs). Elle s'inscrit dans une théorie strictement plus forte que l'arithmétique : un mathématicien peu regardant invoquera la théorie des ensembles de Zermelo-Frenkel mais la théorie constructive des types, conceptuellement plus économique, devrait suffire.

En fait, le théorème de Kirby-Paris [11], qui date de 1982, bien plus difficile que celui de Goodstein, exprime qu'*on ne peut pas prouver la terminaison de l'algorithme de Goodstein avec les seuls axiomes de Peano*. Les axiomes de Peano donnent une description des entiers qui rend compte des propriétés habituelles de l'arithmétique. Il faut constater que la terminaison de l'algorithme de Goodstein est une assertion que l'on peut *exprimer* dans l'axiomatique de Peano.

Il est donc singulier qu'il faille sortir de la théorie de Peano pour la démontrer. Le théorème de Kirby-Paris signifie donc que la théorie de Peano est trop pauvre pour démontrer la terminaison ; en revanche, si on se place dans une théorie assez puissante pour prouver la terminaison de l'algorithme de Goodstein, qui exprime un phénomène de l'arithmétique élémentaire, on interprète le théorème de Kirby-Paris comme un signe que les axiomes de Peano ne rendent pas compte de toutes les caractéristiques des entiers naïfs, dont ils ne constituent ou ne permettent qu'une description partielle.

Rappelons que d'après le théorème d'incomplétude de Gödel, il existe dans toute théorie au moins aussi compliquée que l'arithmétique des assertions qui ne sont ni démontrables, ni réfutables. La preuve de Gödel consistait, en quelque sorte, à coder le paradoxe du menteur (« cette phrase est fausse ») dans la théorie en question. Le théorème de Kirby-Paris donne une assertion explicite qui n'est pas démontrable avec les seuls axiomes de Peano : c'est, à cet égard, une preuve plus directe que la preuve originale du théorème de Gödel.

Conclusion partielle

Dans ce qui précède, en s'intéressant aux propriétés de terminaison et de correction, on a pu démontrer que la plupart des algorithmes présentés donnaient bien le résultat attendu en un nombre fini d'étapes. Ecrire la plupart de ces preuves a mis en jeu des mathématiques significatives mais accessibles.

D'un point de vue mathématique, c'est suffisant pour démontrer les résultats, mais, d'un point de vue pratique est bien loin de permettre le calcul effectif souhaité. En effet, nous ne nous sommes pas intéressés au nombre d'opérations nécessaires à la réalisation de l'algorithme. Le temps d'exécution est fini, certes, mais il dépasse peut-être l'âge de l'univers ! C'est cet aspect que nous nous proposons de développer dans la suite, en présentant et en discutant la notion de complexité.

Complexité d'un algorithme

Sans entrer dans des définitions trop générales, il faut préciser de quoi on veut parler. L'objectif d'un calcul de complexité, c'est d'estimer *a priori* le temps que va mettre un algorithme pour se dérouler. L'algorithmique ne s'intéresse pas à l'implémentation sur une machine particulière, et deux processeurs différents fonctionnent à des fréquences différentes et un processeur donné

11 Kirby, Laurie; Paris, Jeff: Accessible independence results for Peano arithmetic. *Bull. London Math. Soc.* **14** (1982), no. 4, 285—293.

peut avoir plusieurs tâches à accomplir simultanément, etc. Autant dire que l'on n'obtient jamais une réponse en secondes.

Opérations élémentaires

L'idée est plutôt de débiter un algorithme en « opérations élémentaires » et d'estimer le nombre de ces opérations ; sous l'hypothèse que chaque opération prend un temps constant, nous obtiendrons une estimation du temps nécessaire à une constante multiplicative près. Reste que connaître une constante à une constante près, cela n'a évidemment aucun intérêt : il s'agit d'estimer le nombre d'opérations élémentaires en fonction des données initiales, ou plutôt de la *taille* des données initiales.

Un résultat typique, c'est « l'algorithme de tri par tas est en O(n.log(n)) ». Cela signifie que pour trier une liste de n items, le nombre de comparaisons entre items de la liste qu'effectue cet algorithme est majoré par une constante fois n.log(n). De même, dire que « l'algorithme du pivot de Gauss est en $O(n^3)$ », c'est dire qu'il existe une constante C telle que le nombre d'opérations arithmétiques portant sur les coefficients d'un système linéaire de n équations à n inconnues sera, pour tout entier n, au plus Cn^3. On peut raffiner et essayer de trouver C aussi petit que possible.

Mais qu'est-ce qu'une opération élémentaire ? Eh bien, cela dépend. Donald Knuth, dans *The Art of Computer Programming*, est allé jusqu'à imaginer un processeur idéal, MIX, et à détailler les opérations élémentaires jusqu'au nombre de cycles du processeur. Il est rare d'aller jusqu'à ce degré de précision… Ce que l'on prend généralement pour opérations élémentaires, ce sont des opérations bien plus élaborées comme les opérations arithmétiques sur les nombres, les comparaisons (pour les algorithmes de tri), les appels à des fonctions, etc. La question est pertinente mais la réponse varie selon les problèmes.

Par exemple, lorsqu'on résout un système linéaire et que l'on compte uniquement les opérations arithmétiques sur les coefficients, c'est pertinent lorsque ceux-ci sont des nombres flottants ou qu'ils appartiennent à un corps fini : dans ce cas, le codage de chaque coefficient occupe une place de taille constante dans la mémoire et on peut estimer que chaque opération prend un temps constant. En revanche, on peut vouloir faire des calculs exacts avec des rationnels, représentés en machine comme quotients de deux entiers de taille arbitraire. Les logiciels de calcul formel, par exemple Xcas, le permettent. Il faut alors prendre garde que les dénominateurs qui apparaissent dans le déroulement de l'algorithme ont tendance à exploser : dans ce cas, comme une opération arithmétique portant sur de « grands » entiers prend plus de temps qu'une opération avec des « petits » entiers, le nombre d'opérations ne donnera pas une estimation du temps de calculs : il n'est donc pas une mesure pertinente de la complexité de l'algorithme.

Taille des données

Il est raisonnable de s'interroger également sur la taille des données que l'on fournit à l'algorithme. Evidemment, cela va encore dépendre du contexte. La taille d'un entier sera typiquement la place nécessaire pour le stocker en mémoire. Si les entiers sont codés en base 2, c'est le nombre de chiffres en base 2 qui compte. Mais dans les calculs de complexité, on travaille à un facteur constant près, même pour les données : comme tous les logarithmes sont proportionnels, la taille d'un nombre est estimée par son logarithme, quelle que soit la base.

On sait que la sécurité de certains systèmes cryptographiques comme RSA repose sur la

difficulté à factoriser un « grand » nombre entier n. Dans la pratique, « grand » signifie maintenant que n est le produit de deux nombres premiers de 100 à 200 chiffres. Remarquons que l'algorithme naïf consistant à essayer tous les facteurs (impairs) ne fonctionne pas : il faudrait faire environ 10^{100} à 10^{200} essais, ce qui à raison de 10^9 essais par seconde demanderait un temps de calcul *beaucoup* plus grand que l'âge de l'univers. L'enjeu est donc d'importance.

Lorsqu'on résout des systèmes linéaires, une mesure raisonnable de leur taille est souvent donnée par le nombre d'équations et d'inconnues. Du moins, c'est raisonnable si la place prise par chaque coefficient en mémoire est constante (« réels », c'est-à-dire nombres flottants en simple ou double précision, ou éléments d'un corps fini). Avec des systèmes à coefficients entiers, il faut aussi tenir compte de la taille des coefficients. L'algèbre linéaire est tout à fait au point en théorie mais elle est devenue un domaine de recherche pour la mise en œuvre de méthodes efficaces sur machine.[12]

Des exemples

Exemple 1 : calcul récursif
des nombres de Fibonacci

Présentation : Cet algorithme constitue un test de performance (benchmark) classique. Il est très amusant de mesurer les temps d'exécution en fonction de l'indice. Des fichiers proposant une implémentation avec Xcas sont disponibles sur le site de *Repères-IREM*.

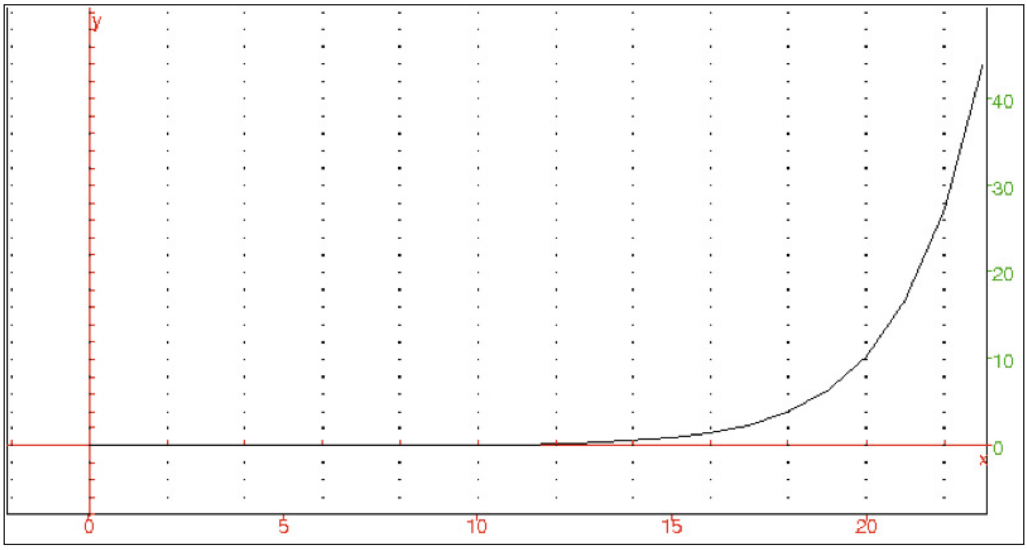
12 Citons le projet LinBox (http://www.linalg.org/), le travail sur les matrices structurées, etc.

Algorithme Fib_rec
Entrée : n entier naturel
Sortie : f(n) entier naturel, le n^e nombre de Fibonacci
Corps :
  si n < 2 **renvoyer** 1
  sinon renvoyer Fib_rec(n–1) + Fib_rec(n–2)
  fin du si

La terminaison et la correction de cet algorithme se prouvent par une récurrence immédiate sur la donnée initiale n. Lorsque n vaut 0 ou 1, l'algorithme renvoie 1, qui est bien la valeur de f(n). Soit n un entier supérieur ou égal à 2. Supposons que l'algorithme se termine et soit correct pour toute valeur de la donnée initiale strictement inférieure à n. Si on fait tourner l'algorithme avec pour valeur initiale 2, la clause « n < 2 » n'est pas satisfaite donc l'algorithme cherche à calculer Fib_rec(n-1) et Fib_rec(n-2). Par hypothèse de récurrence, il va réussir à le faire correctement et en temps fini et va renvoyer la somme de ces deux valeurs : par définition de la suite de Fibonacci, il renvoie bien f(n).

La mise en place de cet algorithme, par exemple avec Xcas, réserve quelques surprises. Pour de petites valeurs de n, disons jusqu'à 15, tout va bien : la réponse est instantanée. Pour n de l'ordre de 20 ou 22, la réponse prend plusieurs secondes pour arriver. Pour n supérieur à 30, elle ne vient apparemment plus du tout. Pourtant, f(30) est inférieur à 2^{30}, nombre comparable à 10^9 puisque 2^{10} vaut à peu près 1000 : le résultat est très loin des limites de la machine !

Mesurons le temps de calcul (en secondes) en fonction de l'indice. Voici le graphe donné par Xcas (Cf. ci-contre). On croirait voir la courbe de la fonction exponentielle. Tâchons de l'expliquer.



Evaluation a priori de la complexité

En observant la fonction Fib_rec, on constate qu'elle met en œuvre deux types d'opérations élémentaires :

— des additions,

— des appels à la fonction Fib_rec elle-même.

Il y a aussi des questions d'allocation de mémoire mais, faute de compétence, on ne rentrera pas là-dedans. En tout état de cause, si on suppose que chaque addition et chaque appel prend un temps constant, mais inconnu, on voit que le temps de calcul T(n) de f(n) par cet algorithme est une combinaison linéaire de S(n) et A(n). On va d'une part calculer ces deux nombres et d'autre part, les comparer aux temps de calculs mesurés.

Pour n entier donné, soit S(n) (respectivement A(n)) le nombre de sommes (respectivement le nombre d'appels) effectuées par l'algorithme pour calculer f(n). On a par exemple :

$$S(0) = S(1) = 0 \text{ et } A(0) = A(1) = 0$$

car pour ces valeurs de n, la réponse n'est pas calculée mais donnée.

Soit n supérieur ou égal à 2. Pour calculer f(n), on effectue

— d'une part, S(n–1) additions pour calculer f(n–1), S(n–2) additions pour calculer f(n–2) et une addition supplémentaire pour calculer f(n–1)+f(n–2) ;

— d'autre part, un appel à Fib_rec pour calculer f(n–1), lequel va engendrer A(n–1) appels à Fib_rec, et un appel à Fib_rec pour calculer f(n–2), lequel va engendrer A(n–2) appels à Fib_rec.

On en déduit :

$$S(n) = S(n-1) + S(n-2) + 1$$

et

$$A(n) = A(n-1) + A(n-2) + 2.$$

On en déduit que les suites auxiliaires définies pour tout n par :

$$s(n) = S(n) + 1 \quad \text{et} \quad a(n) = A(n) + 2$$

satisfont à la même relation de récurrence que la suite de Fibonacci. Admettant, au moins provisoirement, les formules à la Binet, on en déduit que les suites (s(n)) et (a(n)) sont équivalentes [13] à des suites géométriques dont la raison est le nombre d'or. Comme ces suites tendent vers l'infini, c'est aussi vrai des suites (S(n)) et (A(n)).

Ainsi, le nombre d'additions et le nombre d'appels récursifs effectués par la fonction Fib_rec croissent exponentiellement avec la valeur initiale de n.

Mesure des temps de calculs et comparaison

Vérifions que les calculs précédents donnent une explication plausible de l'allure de la courbe ci-dessus. Pour cela, faisons un ajustement exponentiel : représentons le logarithme du temps de calcul de f(n) en fonction de n et la droite de régression linéaire « de y en x » :

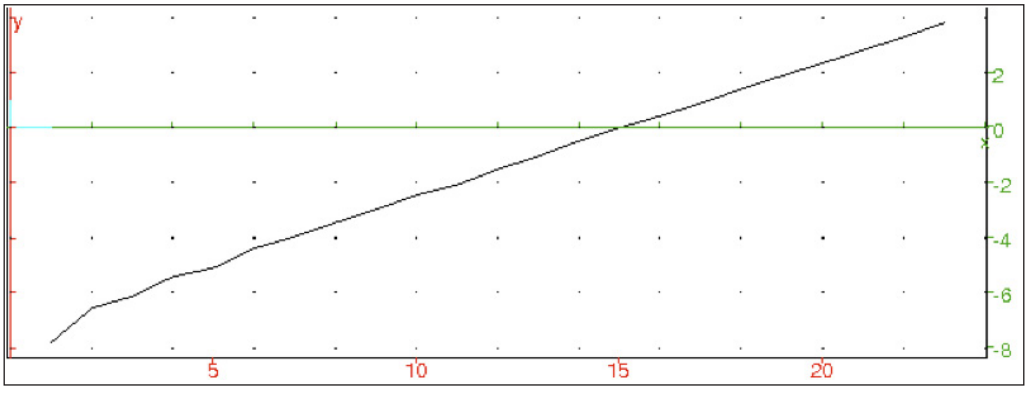
La qualité de l'ajustement est spectaculaire. Les coefficients d'ajustement varient légèrement à chaque exécution, probablement car le temps d'exécution dépend de ce que l'ordinateur fait par ailleurs et que la mesure n'est pas très précise. Dans l'instance représentée ci-dessous, la droite de régression pour les données correspondant à n entre 10 et 23 est :

$$y = 0{,}482943870749\ x - 7{,}32149052099.$$

L'exponentielle du coefficient directeur de la droite est : 1,62, très proche du nombre d'or ! Ces mesures confirment que le calcul de la complexité de l'algorithme récursif par le nombre d'additions et le nombre d'appels donne une image réaliste du calcul de nombres de Fibonacci.

Exemple 2 : calcul itératif des nombres de Fibonacci

Présentation : Comme on va le constater, cet algorithme est beaucoup plus efficace que le précédent. Des fichiers proposant une implémentation avec Xcas sont disponibles sur le site de *Repères-IREM*.



13 Rappelons que deux suites partout non nulles sont équivalentes si leur quotient tend vers 1. Par exemple, la suite (f(n)) est équivalente à la suite géométrique (C ϕ n), où ϕ est le nombre d'or et C est une constante convenable ; les suites (f(n)) et (ϕ f(n-1)) sont équivalentes.

Algorithme Fib_iter
Entrée : n entier naturel
Sortie : f(n) entier naturel, le n^e nombre de Fibonacci
Variables : k, a, b, c entiers
Corps :
a := 1
b := 1
pour k **de** 1 **à** n – 1 **faire**
c := a
a := b
b := a + c
renvoyer b

La terminaison de cet algorithme est évidente. On veut prouver qu'il donne le bon résultat. Lorsque la donnée initiale n vaut 0 ou 1, l'algorithme n'entre pas dans la boucle puisqu'il n'y a aucun entier supérieur ou égal à 1 et inférieur ou égal à n – 1 . Pour n supérieur ou égal à 2, on prouve par une récurrence finie sur k, variant entre 1 et n – 1, que la valeur du couple de variables (a,b) à la sortie du k^e passage dans la boucle est : (a,b) = (f(k), f(k+1)).

Pour k = 1, l'assertion est vraie : à la sortie du premier passage dans la boucle, c vaut 1, a prend la valeur 1 = f(1) et b vaut 1+1 = 2 = f(2). Soit k un entier inférieur ou égal à n – 2, supposons l'assertion vraie au k^e passage : a = f(k) et b = f(k + 1). Après le $(k + 1)^e$ passage dans la boucle, c vaut f(k), a vaut f(k + 1) et b vaut f(k) + f(k + 1) = f(k + 2). Ainsi, après le $(n – 1)^e$ et dernier passage dans la boucle, b vaut f(n – 1 + 1) = f(n). L'algorithme est correct.

Evaluation a priori de la complexité

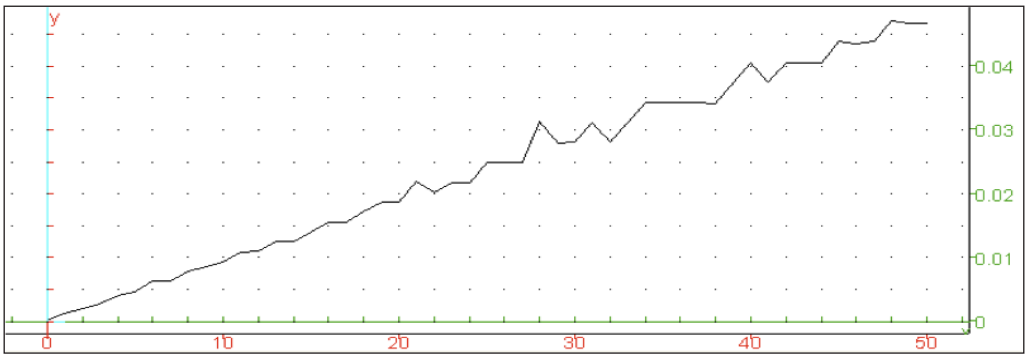
En observant la fonction Fib_iter, on constate qu'elle met en œuvre deux types d'opérations, des affectations et des sommes. Plus précisément, le nombre d'affectations A(n) et le nombre de sommes S(n) effectuées par l'algorithme pour une donnée initiale n > 1 sont :

$$A(n) = 2 + 3\,(n - 1) \quad \text{et} \quad S(n) = n - 1.$$

Sous l'hypothèse que chaque affectation et chaque somme prend un temps constant, on s'attend à ce que le temps de calcul de f(n) par cet algorithme soit à peu près linéaire comme fonction de n. Vérifions-le.

Mesure des temps de calculs et comparaison au calcul a priori

Ici, l'implémentation est réalisée avec Xcas. La première remarque, c'est que le calcul de f(50) est instantané : un net progrès par rapport à l'algorithme précédent… Le graphique ci-dessous



représente le temps de calcul de f(10 × n) en fonction de n. A part les petites oscillations que l'on n'essaiera pas d'expliquer si ce n'est en invoquant l'état de l'ordinateur indépendamment de notre calcul, on constate que le temps de calcul est bien conforme aux attentes, c'est-à-dire linéaire. Le calcul du coefficient directeur de la droite de régression « de y en x » montre que le temps de calcul de f(n) est proche de $n/10^4$ s.

Comparaison des deux algorithmes

Que ce soit du point de vue théorique ou du point de vue pratique, la comparaison des algorithmes récursif et itératif est sans appel : l'un est praticable, l'autre pas vraiment. Cela illustre la différence entre des algorithmes qui fonctionnent « en temps polynômial » et même linéaire, ou « en temps exponentiel ». Comme on va le voir plus loin, on aurait pourtant tort de condamner définitivement les algorithmes récursifs !

Où les masques tombent… Si on teste l'algorithme itératif avec des nombres plus grands (en mesurant le temps de calcul de $f(10^c n)$ pour c constante bien choisie), la courbe ressemble plus à une parabole qu'à une droite (voir le fichier Xcas). C'est naturel si on pense que pour additionner des nombres à d chiffres, il faut faire de l'ordre de d opérations sur les chiffres : ainsi, comme le nombre de chiffres de f(n) est de l'ordre de grandeur de n (c'est à peu près le logarithme dans la base choisie de ϕ^n), on doit faire n opérations avec des nombres de 1 à n chiffres, soit un nombre d'opérations de l'ordre de grandeur de n^2.

Exemple 3 : calcul de puissances (aparté)

Algorithme itératif

On suppose disposer d'une multiplication notée * sur un certain ensemble, par exemple les entiers, les réels ou les matrices. On se donne un élément x de cet ensemble et un entier naturel n : le problème est de calculer la puissance x^n. Faute d'information plus précise, on suppose que chaque multiplication prend un temps constant : la complexité d'un algorithme sera mesurée par le nombre de multiplications.

La première idée consiste à utiliser la relation $x^n = x*x^{n-1}$ pour faire une boucle :

```
Algorithme puiss_iter
Entrée : x élément qu'on sait multiplier, n entier naturel
Sortie : x^n, n-ème puissance de x
Variables : k entier, y
Corps :
  y := x
  pour k de 1 à n – 1 faire
        y := x * y
  renvoyer y
```

D'évidence, cet algorithme demande n – 1 multiplications pour calculer x^n. Ce n'est pas satisfaisant si on pense que la taille d'un entier en est le nombre de chiffres, c'est-à-dire qu'un entier est exponentiel en sa taille. En supposant que n est de l'ordre de 10^{12}, on voit que l'on aura des problèmes. [14]

Algorithme récursif

On remarque alors que si n est pair, n = 2k, il est suffisant de calculer la puissance k^e de x et de l'élever au carré. Si n est impair, n = 2k + 1, on calcule la puissance k^e, on l'élève au carré et on multiplie par x. *Grosso modo*, cela divise le problème par deux : ce n'est pas intéressant en soi (la moitié de l'âge de l'univers,

14 Cela dit, qui veut calculer une puissance 10^{12}-ème ?

c'est toujours trop long !), mais on peut recommencer. Pour calculer la puissance k^e, on applique la même méthode. Cela conduit à l'algorithme récursif suivant :

Algorithme puiss_rec
Entrée : x élément qu'on sait multiplier, n entier naturel
Sortie : x^n
Variable : y
Corps :
  si n = 0 **renvoyer** 1
  sinon
    si n est pair **faire**
      y := puiss_rec(x, n/2)
      renvoyer y * y
    sinon faire
      y := puiss_rec(x, (n–1)/2)
      renvoyer y * y * x

Il est important de ne pas renvoyer puiss_rec(x, n/2) * puiss_rec(x, n/2) car cela forcerait probablement à calculer deux fois la puissance (n/2).

Pour mesurer des temps de calculs sur un ordinateur, il est commode de choisir x entier ; on se retrouve rapidement confronté à des problèmes de taille si on choisit x > 1, ce qui incite à prendre x = 1 pour les tests. Pour compter les opérations, il est commode d'introduire une variable auxiliaire qui sert de compteur. On en arrive à l'algorithme suivant :

Algorithme nb_puiss_rec
Entrée : n entier naturel
Sortie : nombre de multiplications que fait puiss_rec pour calculer x^n
Corps :
  si n = 0 **renvoyer** 0
  sinon
    si n est pair **faire**
      renvoyer nb_puiss_rec(n/2) + 1
    sinon faire
      renvoyer nb_puiss_rec((n–1)/2) + 2

Evaluons le nombre de multiplications M(n) effectuées par l'algorithme puiss_rec(x,n) en fonction de n. Pour un premier encadrement grossier, on note que chaque fois que l'on divise n par 2, on fait 1 ou 2 multiplications de plus. Le nombre total de multiplications est donc compris entre le nombre de fois qu'il faut diviser n par deux pour atteindre 0 ou 1, c'est-à-dire son logarithme en base 2, et le double de ce nombre. On a donc à peu de choses près l'encadrement pour tout n :

$$\log(n) \leq M(n) \leq 2\log(n).$$

Ici, log désigne le logarithme en base 2. Pour être plus précis, on va faire écrire n en base 2 :

$$n = [a_r a_{r-1} \ldots a_1 a_0],$$

où les a_k sont les chiffres de n en base 2. Lorsque n est pair, c'est-à-dire $a_0 = 0$, on a :

$$M(n) = 1 + M(n/2),$$

ce que l'on peut écrire :

$$M([a_r a_{r-1} \ldots a_1 a_0]) = 1 + M([a_r a_{r-1} \ldots a_1]).$$

Lorsque n est impair, c'est-à-dire $a_0 = 1$, on a :

$$M(n) = 2 + M((n-1)/2),$$

ce que l'on peut écrire :

$$M([a_r a_{r-1} \ldots a_1 a_0]) = 2 + M([a_r a_{r-1} \ldots a_1]).$$

On remarque que les deux égalités prennent la même forme :

$$M([a_r a_{r-1} \ldots a_1 a_0]) = 1 + a_0 + M([a_r a_{r-1} \ldots a_1]).$$

Une récurrence immédiate sur r donne alors :

$$M([a_r a_{r-1} \ldots a_1 a_0]) = r + a_r + a_{r-1} + \ldots + a_1 + a_0 .$$

Compte tenu de l'égalité bien connue :

$$r = E(\log(n)),$$

on retrouve l'encadrement initial. On constate que le nombre de multiplications varie selon les chiffres de n en base 2 : il passe pratiquement du double au simple entre $2^r - 1$ et 2^r :

$$M(2^r-1) = M([11\ldots1]) = 2r - 1 \text{ et } M(2^r) = r.$$

Cet algorithme est récursif mais il n'en est pas moins efficace, bien plus que l'algorithme itératif. La raison en est que chaque appel récursif fait intervenir la variable divisée par deux, par opposition à l'algorithme de calcul des nombres de Fibonacci pour lesquels la variable diminuait de 1 ou 2.

Exemple 4 : calcul des nombres de Fibonacci par puissances

Formule de Binet et matrices

Revenons aux nombres de Fibonacci. Pour démontrer la formule de Binet, une méthode standard est de recourir aux matrices : l'intérêt, c'est qu'on ne se contente pas de vérifier une formule tirée d'un chapeau mais qu'on les voit apparaître petit à petit. L'idée est de transformer une récurrence d'ordre 2 portant sur des nombres en une récurrence d'ordre 1 portant sur des vecteurs à deux composantes. On transforme de même les équations différentielles linéaires d'ordre n à une fonction inconnue en systèmes différentiels d'ordre 1 à n inconnues.

On introduit donc la suite de vecteurs (F(n)) définie pour tout entier n par :

$$F(n) = (f(n), f(n + 1)).$$

Soit n entier naturel fixé. Il y a équivalence entre l'égalité

$$f(n + 2) = f(n + 1) + f(n)$$

et le système de deux égalités :

$$f(n+1) = f(n+1)$$

$$f(n+2) = f(n) + f(n+1),$$

ce qui peut s'écrire :

$$F(n+1) = F(n).\ A, \text{ où } A = \quad .$$

Aparté : On reconnaît sous cette forme le même problème que l'on rencontre avec les graphes probabilistes ! Il s'agit de trouver les valeurs propres u et v de la matrice A : si elle en a deux distinctes, on en déduit des combinaisons linéaires des composantes de F(n) qui sont des suites géométriques de raisons u et v. Lorsque A est une matrice stochastique, on sait *a priori* que u = 1 est une valeur propre et que l'autre valeur propre satisfait à $|v| < 1$. La combinaison linéaire correspondant à la valeur propre 1 correspond à l'état stable du graphe probabiliste.

Dans notre cas très précis, on vérifie que les valeurs propres de A, c'est-à-dire les raisons possibles d'une suite géométrique de la forme (a.f(n) + b.f(n + 1)), avec a et b réels, sont le nombre d'or ϕ et son inverse.

On peut écrire un système de la forme :

$$a.f(n) + b.f(n+1) = C\ \phi^n$$

$$c.f(n) + d.f(n+1) = D\ \phi^{-n}$$

où a, b, c, d, C, D sont des constantes bien déterminées et n un entier quelconque. En le résolvant, on trouve les formules de Binet.

Calcul matriciel des nombres de Fibonacci

La relation de récurrence de la suite (F(n)) donne l'expression suivante, valable pour tout n :

$$F(n) = F(0).A^n.$$

On peut en fait démontrer sans trop de peine que l'on a pour tout $n > 0$ (on pose $f(-1) = 0$) :

$$A^n = \begin{pmatrix} f(n-2) & f(n-1) \\ f(n-1) & f(n) \end{pmatrix}.$$

On obtient ainsi une façon de calculer les nombres de Fibonacci par un simple calcul de puissance… de matrices. Vu l'exemple 3, on s'attend à ce que le calcul soit bien plus efficace, et c'est le cas ! En effet, le nombre d'opérations sur les nombres pour multiplier deux matrices 2×2 est fixe (8 multiplications et 4 additions). On obtient ainsi une méthode de calcul du nombre $f(n)$ qui est majoré par une constante fois $\log(n)$. C'est un gain énorme par rapport aux algorithmes récursif et itératif.

On trouve sur le site de *Repères-IREM* une implémentation de l'algorithme correspondant avec Xcas, en faisant semblant que ce logiciel ne connaît pas le calcul matriciel. On constate plusieurs choses :

— d'abord, on peut pousser le calcul pour de bien plus grandes valeurs de n ; par exemple, le calcul de $f(2^{20}+3)$ ne prend que 0,14 s, alors que la taille de ce nombre est gigantesque : plus de 200 000 chiffres ;

— ensuite, le temps de calcul est encore endessous des capacités de mesure alors qu'il prend des secondes entières avec l'algorithme itératif : il faut environ 10 s pour calculer $f(100\,000)$ par l'algorithme itératif et moins de 3 centièmes de secondes avec l'algorithme par puissances ;

— enfin, pour de très grandes valeurs de n, le temps est plus grand que ce que l'on attend : nous interprétons ce phénomène par le fait que pour ces valeurs, la taille des nombres que l'on manipule est gigantesque, si bien que l'hypothèse que les opérations arithmétiques prennent un temps constant devient fausse.

Exemple 5 : résolution de systèmes linéaires par l'algorithme du pivot de Gauss

Evaluation du nombre d'opérations

Considérons pour simplifier un système où il y a autant d'équations que d'inconnues.

Soit $A = (a_{ij})_{i,j=1,\ldots,n}$ la matrice du système. La première étape de l'algorithme de Gauss consiste à manipuler les lignes pour rendre la matrice triangulaire supérieure. Pour cela, on va supposer que $a_{11} \neq 0$ — sinon, $n-1$ comparaisons et une permutation de deux lignes permettent de remplacer le pivot a_{11} par le coefficient de valeur absolue maximale sur la première colonne. Soit i un entier compris entre 2 à n. On remplace la i^e ligne L_i du système par $L_i - c_i L_1$, où $c_i = a_{i1}/a_{11}$. Pour cela, on effectue une division pour calculer c_i, n multiplications et autant d'additions pour la matrice et le second membre, soit $(n + 1)$ multiplications et n additions. Comme il y a $n - 1$ lignes, cela fait, pour cette étape, $n^2 - 1$ multiplications et $n^2 - n$ additions. On s'est ainsi ramené à un système de taille $(n - 1) \times (n - 1)$. Pour cette première étape, on trouve donc :

— nombre de multiplications :

$$\sum_{k=1}^{n}(k^2-1) = \frac{n(n-1)(2n+5)}{6} \approx \frac{n^3}{3}$$

— nombre d'additions :

$$\sum_{k=1}^{n}k(k-1) = \frac{n(n-1)(n+1)}{3} \approx \frac{n^3}{3}.$$

On a à présent un système triangulaire à résoudre. La dernière variable s'obtient avec une division. L'avant-dernière s'obtient avec une multiplication, une soustraction et une division. Pour $k \geq 1$, le calcul de la variable numéro $n - k$ nécessite $k - 1$ multiplications, $k - 1$ addi-

tions et une division, soit, en tout, k multiplications et k – 1 additions. Dans cette étape, on trouve donc :

— nombre de multiplications :

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2} \approx \frac{n^2}{2}$$

— nombre d'additions :

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

En ajoutant le nombre d'opérations des deux étapes et en supposant n grand, on constate que l'algorithme de Gauss demande environ $n^3/3$ additions et multiplications ; en simplifiant un peu : l'algorithme de Gauss pour une matrice $n \times n$ demande $O(n^3)$ opérations, c'est-à-dire, au plus une constante fois n^3 opérations.

On peut remarquer que l'on a bien fait de négliger les comparaisons car il en faut au maximum n – 1 pour la première colonne, n – 2 pour la deuxième, etc., soit $O(n^2)$ en tout. Quant aux permutations de lignes, il y en a O(n) au maximum. Tout cela est négligeable devant n^3.

Un exemple d'application

On tombe sur des systèmes de grande taille quand on veut résoudre numériquement des équations aux dérivées partielles. Imaginons que l'on veut déterminer la pression d'un fluide dans un cube. On divise chaque arête du cube en 100 intervalles, ce qui ne semble pas une précision extravagante mais produit 10^6 petits cubes : la pression dans chacun de ces petits cubes est une inconnue, d'où, dans un contexte de calcul scientifique assez banal, un système d'un million d'équations à un million d'inconnues. Si on applique l'algorithme de Gauss, il faudra donc de l'ordre de 10^{18} opérations ! Si le nombre d'opérations par seconde est de l'ordre du milliard, il faut envisager un milliard de secondes par système : plusieurs années ! C'est inenvisageable : il n'est même pas utile de programmer le calcul de cette façon...

Exemple 6 : complexité de l'algorithme d'Euclide

On cherche à majorer le nombre de divisions euclidiennes qu'effectue l'algorithme d'Euclide en fonction des données.

Théorème de Lamé

Soient a_0 et a_1 les nombres dont on cherche le pgcd. On les suppose strictement positifs et « dans le bon ordre » : $a_0 \leq a_1$. L'algorithme définit deux suites finies (a_k) et (q_k) de proche en proche par :

$$\begin{aligned}
&a_0 = a_1 q_1 + a_2, && 0 \leq a_2 < a_1\\
&a_1 = a_2 q_2 + a_3, && 0 \leq a_3 < a_2\\
&\dots\\
&a_{r-2} = a_{r-1} q_{r-1} + a_r, && 0 \leq a_r < a_{r-1}\\
&a_{r-1} = a_r q_r + 0, && a_{r+1} = 0.
\end{aligned}$$

On appellera r le nombre d'étapes. C'est le nombre de lignes du tableau, donc le nombre de divisions euclidiennes à effectuer. L'idée, c'est que le nombre d'étapes est maximal lorsque la suite (a_k) décroît le moins possible, c'est-à-dire lorsque les quotients successifs (q_k) sont aussi petits que possible.

On a dans ce cas :

$$a_{k+2} = a_k - k+1 \ ,$$

ce qui ressemble fort à la suite de Fibonacci. Pour formaliser cette idée, on renumérote les (a_k) en commençant par la fin, ce qui revient à poser, pour k entre 1 et r + 1 :

$$u_k = a_{r+1-k} \ .$$

Pour i entier compris entre 1 et r, l'égalité

$$a_{i-1} = q_i a_i + a_{i+1}$$

s'écrit, pour k entier compris entre 1 et r :

$$u_{k+1} = q_{r+1-k}\, u_k + u_{k-1}.$$

La remarque clé, c'est que tous les q_{r+1-k} valent au moins 1, ce qui donne pour tout k :

$$u_{k+1} \geq u_k + u_{k-1}.$$

On est conduit à comparer (u_k) à la suite finie (g(k)) définie par :

$$g(0) = 0,\ g(1) = 1,\ \ g(k+1) = g(k) + g(k-1)$$

pour tout k, qui est bien sûr la suite de Fibonacci à un décalage près : f(k) = g(k + 1) pour tout k. On a : $u_0 = 0 = g(0)$ et $u_1 \geq g(1)$, d'où par une récurrence immédiate :

$$u_k \geq g(k) \text{ pour tout } k \leq r+1.$$

En particulier, $u_{r+1} \geq g(r+1)$, soit $a_0 \geq f(r)$.

En d'autres termes, le nombre d'étapes de l'algorithme d'Euclide r est, au pire, l'indice du plus grand nombre de Fibonacci inférieur au plus grand des deux entiers. Il y a égalité si on part des entiers $a_0 = f(r)$ et $a_1 = f(r-1)$. D'après la formule de Binet, on en déduit que

$$r \leq \log_{\phi}(a_0) + \text{Cste}.$$

Ceci traduit que l'algorithme d'Euclide est efficace, puisqu'il est linéaire en la taille des données (leur logarithme). Cette majoration du nombre d'étapes de l'algorithme d'Euclide est parfois une très mauvaise estimation : si par exemple a_0 est un multiple de a_1, il suffit d'une seule division euclidienne ! On est dans une situation où la complexité « dans le cas le meilleur » est bien plus favorable que « dans le cas le pire ». Il est plus intéressant, si on est amené à calculer de nombreux pgcd, de calculer un nombre d'étapes « en moyenne », disons, sur les paires de nombres inférieurs à un entier N donné. Ce que l'on trouve, mais c'est vraiment beaucoup plus difficile que le théorème de Lamé que l'on vient de démontrer, c'est que le nombre moyens de divisions est équivalent à C.log(N) pour C une constante convenable. Voir par exemple *The Art of ComputerProgramming* de Donald Knuth, volume 2.

Variante : On peut trouver une estimation plus rapide à l'aide d'une astuce. Au lieu de prendre le reste d'une division euclidienne de a par b entre 0 et b – 1, on peut s'arranger pour le prendre entre – b/2 et b/2 : si le reste r est supérieur à b/2, on le remplace par r' = r – b. De la sorte, on est sûr qu'à chaque étape, le reste est au moins divisé par deux. On obtient ainsi un nombre de divisions euclidiennes qui est au pire le logarithme en base 2 des données initiales : c'est un petit gain.

Conclusion

Dans la perspective de l'enseignement de l'algorithmique au lycée, ces quelques exemples se veulent des pistes pour construire des cours présentant les questions fondamentales de l'algorithmique. Est-ce qu'un algorithme s'arrête ? Est-ce qu'il donne le résultat que l'on attend ? Est-ce dans un temps raisonnable ? Ces exemples montrent qu'il est possible, pour aborder ces sujets, de partir d'algorithmes simples, souvent naturellement dans les programmes et facilement programmables sur des machines ; ils peuvent permettre d'approcher des méthodes faisant appel aux connaissances mathématiques des élèves et montrant tant d'un point de vue théorique qu'expérimental comment représenter la complexité, la correction et les preuves d'algorithmes. Qui plus est, les problèmes posés par l'algorithmique débouchent naturellement sur des points à traiter dans le cœur du programme : récurrence, majorations, logarithme, suites satisfaisant à des relations de récurrence non triviales, *etc*.

Bibliographie sommaire

BO n°30 du 3 juillet 2009 : http://media.education.gouv.fr/file/30/52/3/programme_mathematiques_seconde_65523.pdf

JO du 28 août 2010 portant sur les programmes de première :

http://www.legifrance.gouv.fr/affichTexte.do?cidTexte=JORFTEXT000022746962

et

http://www.legifrance.gouv.fr/affichTexte.do;jsessionid=7529E063D436BC2F3ABC9386F7035635.tpdjo08v_1?cidTexte=JORFTEXT000022746934&dateTexte=&oldAction=rechJO&categorieLien=id

ABDELJAOUED, Jounaïdi, LOMBARDI, Henri (2004) : *Méthodes matricielles: introduction à la complexité algébrique*. Mathématiques & Applications 42. Springer-Verlag, Berlin.

CORMEN Thomas, LEISERSON Charles, RIVEST Ronald, STEIN Clifford (2010, 3e édition) : *Algorithmique*, Cours avec 957 exercices et 158 problèmes, éditions Dunod.

EXPRIME, *Expérimenter des problèmes de recherche innovants à l'école*, (2010) cédérom édité par l'INRP.

HOROWITZ, E., SAHNI, S. (1978) *Fondamentals of Computer Algorithms*, Editions Pitman

KNUTH Donald (1998) : *The Art of Computer Programming*, Addison-Wesley.

MINES, Ray, RICHMAN, Fred, RUITENBURG, Wim (1988) : *A course in constructive algebra*. Universitext. Springer-Verlag, New York.

NGUYỄN Chí Thành (2005) : *Etude didactique de l'introduction d'éléments d'algorithmique et de programmation dans l'enseignement mathématique secondaire à l'aide de la calculatrice*. Thèse de doctorat, Université Joseph-Fourier-École normale supérieurs de Hanoï

ROBERT Yves (2005) : *Cours d'algorithmique*, notes de cours donné à l'École normale supérieure de Lyon (certaines parties sont très accessibles, d'autres sont plus sophistiquées) : http://graal.ens-lyon.fr/~yrobert/algo/poly-algo.ps