Chapitre 3

Autour du PGCD de deux entiers

La notion de pgcd de deux entiers est essentielle en arithmétique. L'algorithme que l'on met en œuvre pour le calculer est un des plus anciens de toute l'histoire des mathématiques, puisqu'on le trouve déjà exposé dans les *Éléments* d'Euclide, et aussi l'un des plus performants. Son étude s'impose donc. Nous verrons aussi comment écrire sur la TI-Nspire le calcul des coefficients de Bézout.

Sommaire

1. pgcd de deux entiers

1.1 *Un algorithme classique*

Le principe de cet algorithme est élémentaire. Il est basé sur l'idée simple que le pgcd de deux entiers *a* et *b* est le même que le pgcd de *b* et de *r*, où *r* est le reste de la division euclidienne de *a* par *b* (obtenu sur notre calculatrice par l'instruction **mod(a,b)**).

Rien n'empêche de réitérer le processus, tant que la division est possible.

Avec les notations suivantes, que nous utiliserons tout au long de ce chapitre, on peut détailler les différentes étapes :

division de a par b : $a = bq_0 + r_0$ avec $0 \leqslant r_0 < b$

division de b par r_0 : $b = r_0q_1 + r_1$ avec $0 \leqslant r_1 < r_0$

division de r_0 par r_1 : $r_0 = r_1q_2 + r_2$ avec $0 \leqslant r_2 < r_1$

…

division de r_{i-1} par r_i : $r_{i-1} = r_iq_{i+1} + r_{i+1}$ avec $0 \leqslant r_{i+1} < r_i$

…

division de r_{n-2} par r_{n-1} : $r_{n-2} = r_{n-1}q_n + r_n$ avec $0 \leqslant r_n < r_{n-1}$

division de r_{n-1} par r_n : $r_{n-1} = r_nq_{n+1} + 0$

Comme les restes successifs forment une suite d'entiers naturels strictement décroissante, on arrive nécessairement à un reste nul et le processus s'arrête. On peut alors écrire :

$$\text{pgcd}(a,b) = \text{pgcd}(b,r_0) = \text{pgcd}(r_0,r_1) = \ldots = \text{pgcd}(r_{n-1},r_n) = r_{n-1}$$

En d'autres termes, le pgcd de *a* et de *b* est le dernier reste non nul dans cette suite de divisions successives.

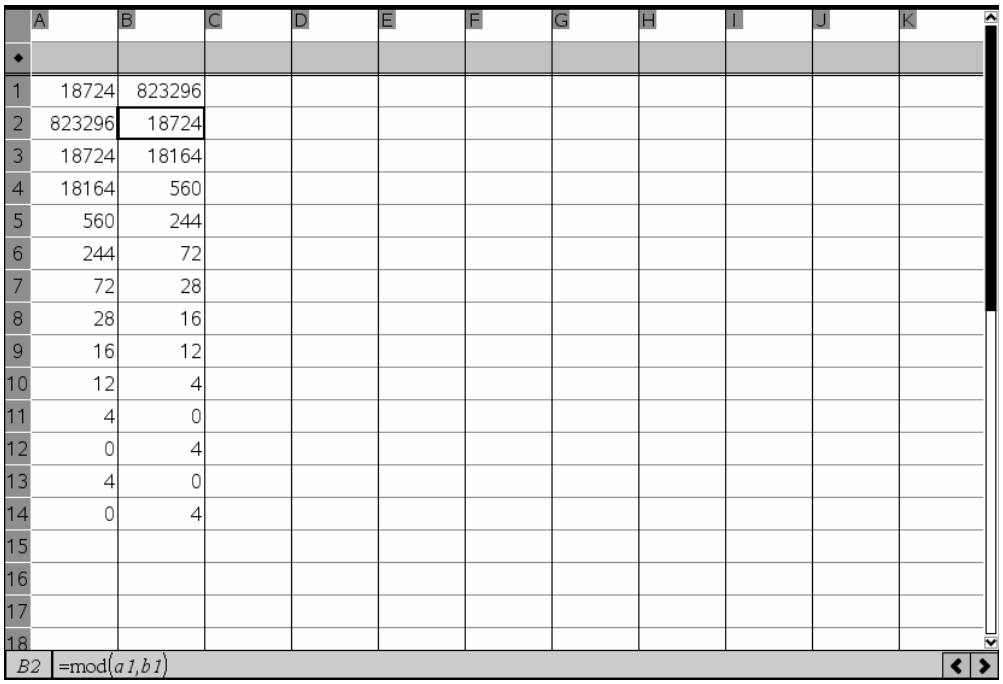
On sait que la TI-Nspire dispose d'une fonction **gcd** particulièrement puissante (**g**reatest **c**ommon **d**ivisor in english), permettant le calcul du pgcd de deux entiers. À tel point qu'on pourrait douter de l'intérêt d'en écrire soi-même une, nécessairement moins rapide.

Mais c'est oublier que l'algorithme d'Euclide est *l'archétype* des algorithmes, sans doute un des plus anciens, un des plus efficaces aussi. D'une simplicité extrême, il renvoie pour autant un résultat non trivial et absolument fondamental en arithmétique. Autant de raisons qui font qu'un jour ou l'autre, il faut l'écrire…

1.2 *L'algorithme sur le tableur*

- La démarche décrite précédemment se traduit très simplement dans une feuille de calcul. Les nombres *a* et *b* sont mis dans les cellules A1 et B1 (18 724 et 823 296 dans notre exemple) ; puis dans A2, on tape **=b1** et dans B2, **=mod(a1,b1).** On recopie les deux cellules A2 et B2 vers le bas sur quelques lignes**.**

Le pgcd est ici 4, le dernier reste non nul. On remarque qu'il est obtenu en assez peu d'étapes, malgré la taille des nombres qui interviennent.



	A	B	C	D	E	F	G	H	I	J	K
1	18724	823296									
2	823296	18724									
3	18724	18164									
4	18164	560									
5	560	244									
6	244	72									
7	72	28									
8	28	16									
9	16	12									
10	12	4									
11	4	0									
12	0	4									
13	4	0									
14	0	4									
15											
16											
17											
18											

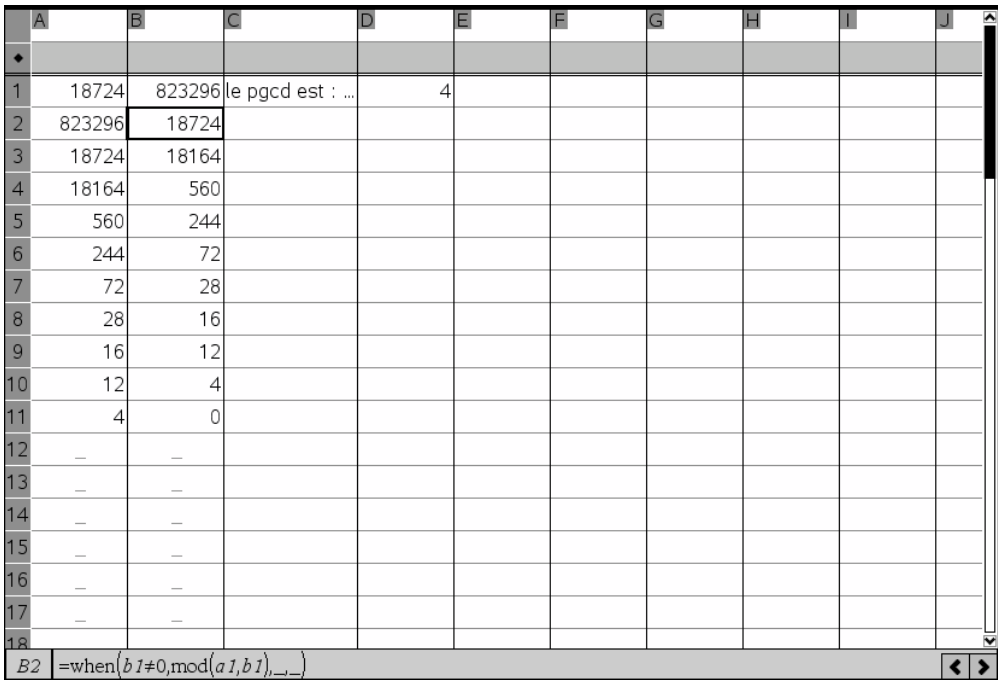
B2 =mod(a1,b1)

- Pour faire une feuille de calcul propre, on peut conditionner la copie au fait que B1 soit non nul et utiliser **void** dans le cas contraire. On peut par exemple saisir :

When(b1≠0,b1,void, void) en A2,

When(b1≠0,mod(a1,b1),void, void) en B2,

et recopier vers le bas sur une trentaine de lignes.



	A	B	C	D	E	F	G	H	I	J
1	18724	823296	le pgcd est : ...	4						
2	823296	18724								
3	18724	18164								
4	18164	560								
5	560	244								
6	244	72								
7	72	28								
8	28	16								
9	16	12								
10	12	4								
11	4	0								
12	_	_								
13	_	_								
14	_	_								
15	_	_								
16	_	_								
17	_	_								
18										

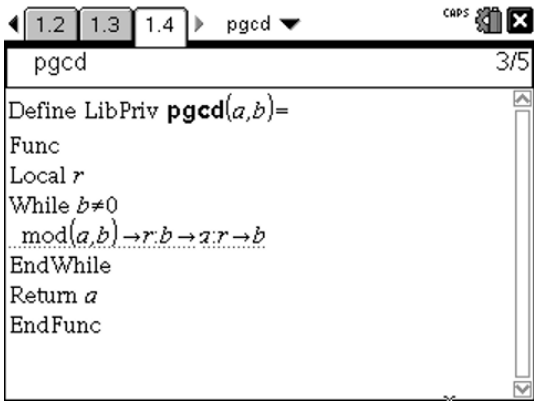
B2 =when(b1≠0,mod(a1,b1),_,_)

Un petit trait signale la cellule vide lorsque le calcul n'est plus effectué, soit parce que la cellule de référence contient 0, soit parce qu'elle est vide.

En prime, on peut faire apparaître pgcd en D1, qui n'est autre que le minimum de la colonne A(**=min(a[])**)… les **void** ne sont pas comptés !

1.3 *Une fonction pgcd*

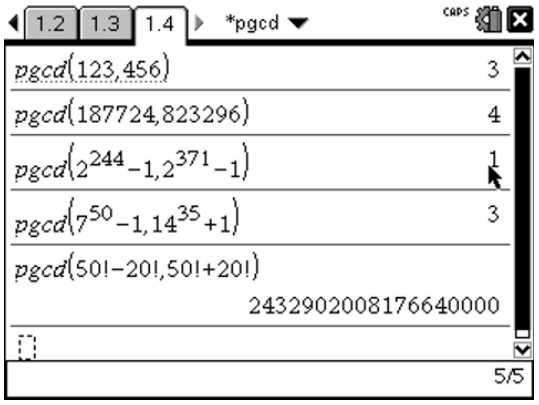
- Passons maintenant à l'écriture d'une fonction **pgcd**. La feuille de calcul que nous venons d'écrire est déjà la mise en œuvre de l'algorithme : la succession des opérations précédentes sur chacune des lignes peut se traduire dans une fonction par une boucle **While**, parce qu'on ne sait pas à l'avance combien d'étapes seront nécessaires.



Quelques remarques de bon sens avant de commencer… Contrairement à ce que l'on pourrait penser, il n'est pas nécessaire de supposer au préalable que *a* est plus grand que *b* pour calculer le pgcd de *a* et *b*. Dans le cas où *a* est strictement inférieur à *b* la première division remet les nombres dans l'ordre : c'est d'ailleurs ce qui se passe sur l'exemple traité au tableur.
On peut aussi éviter de définir de trop nombreuses variables locales : une seule suffit, *r*, pour mémoriser les restes successifs.

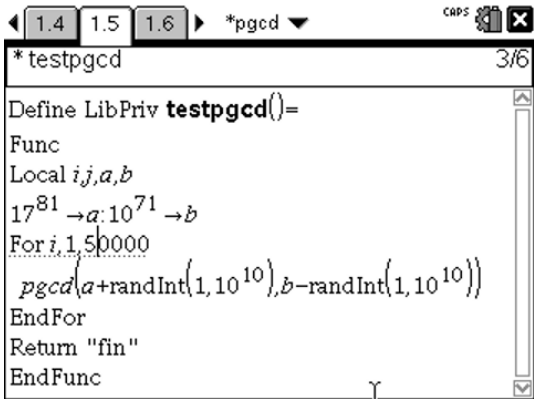
On sort donc de la boucle **While** dès que *b* est nul. Dans ce cas, le pgcd cherché est *a* (car pgcd(*a*,0) = *a*). À l'intérieur de la boucle, il faut être vigilant aux affectations qui sont faites, et notamment à leur ordre. Si la première instruction à exécuter est clairement le calcul de **mod(a,b)** stocké dans *r*, il faut ensuite mettre *b* dans *a*, puis *r* dans *b*, et non l'inverse.

Les résultats sont obtenus quasi immédiatement, même si les nombres en jeu sont impressionnants et dépassent la capacité des calculatrices usuelles.



On sait (*voir annexe*) que le nombre maximal d'étapes est majoré par 5*k*, où *k* est le nombre de chiffres du plus *petit* des entiers *a* et *b* : ainsi avec un entier d'une vingtaine de chiffres, *au pire* on effectuera une centaine de divisions... peut-être moins… ce que la TI-nspire fait très rapidement, beaucoup plus rapidement qu'une Voyage 200…

Comparons notre pgcd avec la fonction **gcd** de la calculatrice, au moyen de la fonction suivante qui demande le calcul du pgcd de deux nombres 50 000 fois. Une première fois, on utilise notre pgcd et une deuxième fois le pgcd de la calculatrice…



Montre en main, avec **pgcd**, cette fonction demande 1 minutes 45 s sur un ordinateur portable type Acer ; en remplaçant **pgcd** par **gcd** dans l'instruction de la boucle **For**, elle s'exécute en seulement 5 secondes !

Bilan : on s'en doutait mais la fonction **gcd** de la calculatrice est largement plus performante.

- **Une tentative d'amélioration…**

Notre fonction, au moins en théorie, peut être légèrement améliorée, avec la remarque suivante : le pgcd de *b* et de *r* est aussi le même que le pgcd de *b* et de *b* – *r*.
En remplaçant *r* par le plus petit des nombres *r* et *b* – *r*, la suite des restes va décroître un peu plus vite vers 0… ce que montre la feuille de calcul suivante :

	A	B	C	D	E	F	G	H	I	J	K
◆											
1	754321	823296	1	754321	823296	1					
2	823296	754321	16	823296	68975	9					
3	754321	68975		68975	4404						
4	68975	64571		4404	1489						
5	64571	4404		1489	63						
6	4404	2915		63	23						
7	2915	1489		23	6						
8	1489	1426		6	1						
9	1426	63		1	0						
10	63	40		_	_						
11	40	23		_	_						
12	23	17		_	_						
13	17	6		_	_						
14	6	5		_	_						
15	5	1		_	_						
16	1	0		_	_						
17	_	_		_	_						
18											

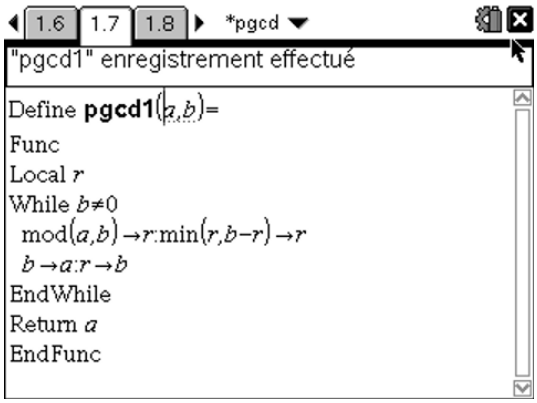
E2 =when(e1≠0,min(mod(d1,e1),e1−mod(d1,e1)),_,_)

Les deux premières colonnes, comme précédemment, donnent la valeur du pgcd en C1 et le nombre d'étapes en C2 (**=dim(delvoid(a[]))**. Les colonnes C et D permettent un calcul parallèle du même pgcd, mais dans E2 a été entré :

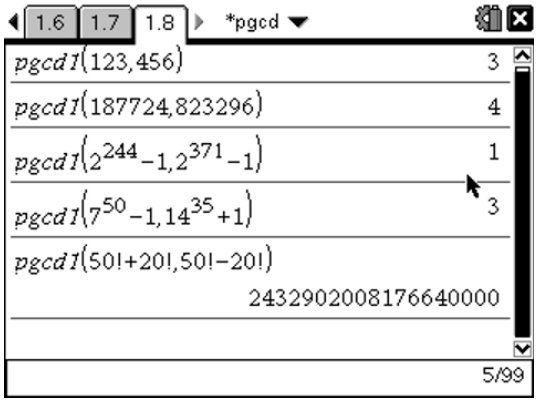
=when(e1≠0,min(mod(d1,e1),e1−mod(d1,e1)),void,void)

On constate qu'il faut en effet moins de valeurs avec cette deuxième méthode : en regardant les résultats obtenus, on observe qu'on saute quelques étapes…

On peut donc espérer que la fonction **pgcd1** soit un chouia plus rapide…



Cette nouvelle fonction renvoie bien les mêmes résultats que précédemment :



Mais quand on teste sa rapidité avec **testpgcd**, les résultats sont finalement décevants : aussi étonnant que cela puisse paraître, elle demande à peine moins de temps qu'avec la fonction **pgcd :** 2 minutes 5 contre 2 minutes 10 (sur l'ordinateur)**.** Bref, le gain n'est pas spectaculaire**.** Faire vite certes, mais encore faut-il que le code reste simple !

1.4 *La méthode des différences successives*

Incontournable, pour des raisons historiques d'une part – la méthode est décrite dans les *Éléments* d'Euclide –, de simplicité d'autre part – elle s'appuie sur des différences de nombres.

Pour le plaisir, citons la proposition I du livre VII des *Éléments* d'Euclide, le premier des trois livres consacrés à l'arithmétique :

> *Proposition I*
> *Deux nombres inégaux étant proposés, le plus petit étant toujours retranché du plus grand, si le reste ne mesure*[1] *celui qui est avant lui que lorsque l'on a pris l'unité, les nombres proposés seront premiers entre eux.*

Et la proposition II du livre VII prouve que cette méthode des différences successives conduit en fait au PGCD des entiers initiaux.

En langage moderne, on s'appuie sur le résultat suivant :

> Si a et b sont deux entiers naturels tels que $a > b$, alors on a :
> $\text{pgcd}(a,b) = \text{pgcd}(b,a-b)$.

La mise en œuvre est élémentaire, que ce soit dans une feuille de calcul ou dans une fonction.

[1] Dire que b mesure a signifie que b divise a. Ainsi 2 mesure 6. L'interprétation « géométrique » presque évidente : en 6 combien de fois 2 ? Exactement 3 fois…

On remplace la recherche du pgcd de *a* et de *b* (*a* > *b*) par celle du couple obtenu en conservant *b* (le plus petit nombre) et en remplaçant *a* (le plus grand nombre) par la différence des deux nombres, *a* – *b*. On réitère le procédé jusqu'à arriver à un des nombres égal à 0 : l'autre nombre est alors le pgcd de *a* et *b*.
La question est de savoir si l'on est sûr que l'un des deux nombres soit effectivement à un moment égal à 0. Or, par construction, le plus grand des deux nombres décroît au cours des différentes étapes. Il décroît même strictement tant que l'autre nombre n'est pas nul. Mais cette stricte décroissance d'entiers naturels ne peut pas se poursuivre indéfiniment : il arrive donc un moment où le plus petit des deux nombres vaut 0 et où le plus grand des deux nombres garde la même valeur. Quand c'est le cas, le nombre non nul est le pgcd. À l'étape d'avant les deux nombres étaient nécessairement égaux…

Sur le tableur, dans la cellule A2, tant que **min(a1,b1)** n'est pas nul, on y place **=min(a1,b1)** ; dans la cellule voisine B2, tant que A2 n'est pas vide (**void**), on y place **=abs(a1 – b1).** On a bien remplacé le plus grand des deux nombres par la différence des deux**.** Le tout est recopié vers le bas autant que nécessaire… Le pgcd est le minimum de la colonne A**.**

	A	B	C	D	E	F	G	H	I	J	K
◆											
1	2450	2114	14								
2	2114	336									
3	336	1778									
4	336	1442									
5	336	1106									
6	336	770									
7	336	434									
8	336	98									
9	98	238									
10	98	140									
11	98	42									
12	42	56									
13	42	14									
14	14	28									
15	14	14									
16	14	0									
17	_	_									
18											

A2 =when(min(a1,b1)≠0,min(a1,b1),_,_)

Une remarque qui montre le lien entre cette méthode et l'algorithme d'Euclide vu précédemment, faisant intervenir des divisions. À partir de la ligne 2, on doit retirer un certain nombre de fois 336 de 2114. Les soustractions successives des lignes 2 à 8 sont fastidieuses ; en une fois, il est plus simple de faire la division de 2114 par 336, et d'en récupérer le reste :

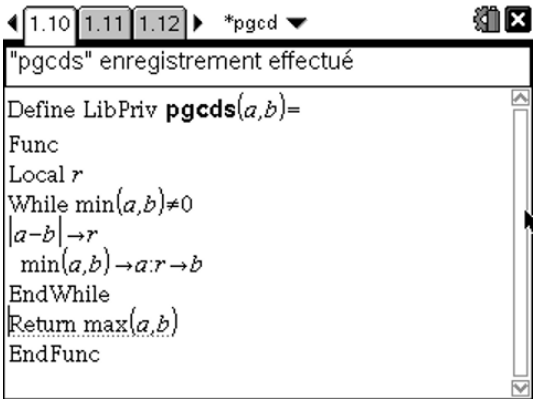
$$2\,114 = 6 \times 336 + 98$$

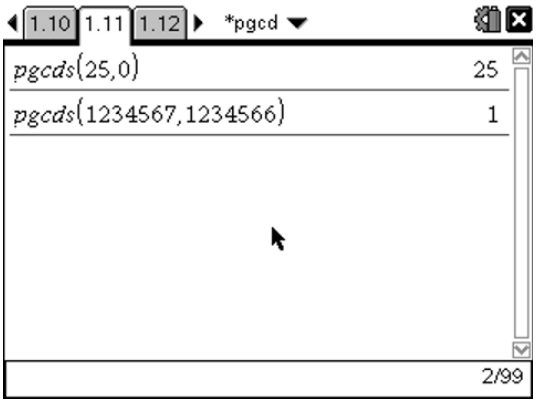
d'où l'on tire

$$2\,114 - 6 \times 336 = 98.$$

Avec la division, qui n'est donc comme on le sait qu'une soustraction répétée, on passe donc directement de la ligne 2 (2114-336) à la ligne 8 (336-98).

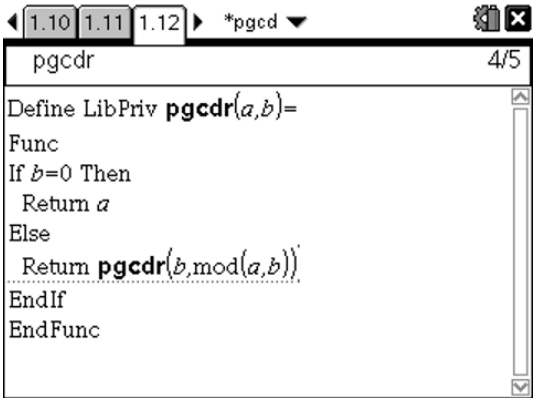
Il est facile d'écrire un programme de soustractions successives mais il ne faut pas en attendre des performances spectaculaires… Il est même particulièrement lent, et l'on vient d'expliquer pourquoi. Le principe est le même que celui de la feuille de calcul : on place **min(a,b)** dans *a* et **abs(a−b)** dans *b*. Lorsque le minimum atteint la valeur 0, le pgcd est égal à l'autre valeur.

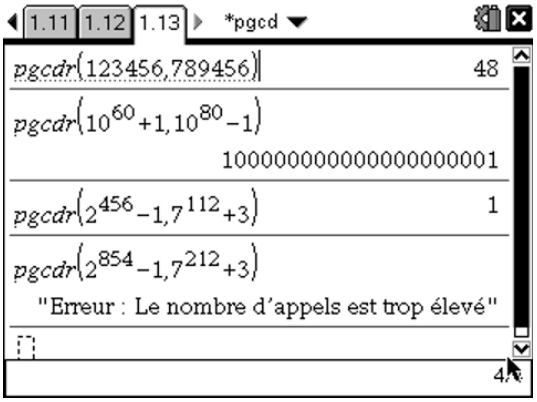




1.5 *Un pgcd récursif*

Juste pour le plaisir, car ce que nous allons écrire sera peu performant… La TI-Nspire, comme les modèles de calculatrices qui l'ont précédée, autorise la récursivité. En d'autres termes, on peut donc écrire un programme, ou une fonction, qui s'appelle lui même. Voici par exemple un pgcd récursif :





Il y a là de quoi rester incrédule : comment cette fonction peut-elle effectivement renvoyer le pgcd de *a* et de *b* alors qu'en apparence, rien n'a été écrit, sinon la définition du dit pgcd ?

En y regardant de plus près, on constate que la fonction **pgcdr** s'appelle en fait *elle-même* après le **Else** : c'est ce qui caractérise, comme on l'a rappelé, la récursivité.

Que se passe-t-il donc pour le calcul du pgcd de $a = 72$ ou $b = 45$ par exemple ?

> Comme $b = 45$ n'est pas nul, elle renvoie **pgcdr(45,mod(72,45))=pgcdr(45,27).** Une deuxième fois, la fonction **pgcdr** est appelée c'est-à-dire que deux versions de cette fonction vont donc être exécutées en même temps.
>
> Comme $b = 27$, elle renvoie **pgcdr(27,mod(45,27))=pgcdr(27,18).** Une troisième fois, la fonction **pgcdr** est appelée.
>
> Comme $b = 18$, elle renvoie **pgcdr(18,mod(27,18))=pgcdr(18,9).** Une quatrième fois, la fonction **pgcdr** est appelée.
>
> Comme $b = 9$, elle renvoie **pgcdr(9,mod(18,9))=pgcdr(9,0).** Une cinquième fois, la fonction **pgcdr** est appelée et renvoie comme réponse 9.

Il est absolument essentiel qu'à un moment, la condition du test soit vérifiée : sinon, la fonction serait appelée indéfiniment, jusqu'à saturation de la mémoire. Saturation de la mémoire qui peut aussi se produire dès que le nombre d'appels est trop grand : c'est le cas du dernier exemple de l'écran précédent.

On peut penser qu'une fonction récursive, par les ressources qu'elle mobilise, sera en général moins rapide qu'une fonction écrite classiquement : c'est effectivement le cas[2], mais de façon peu sensible pour ce programme. La récursivité a cependant pour elle l'élégance et la simplicité : le fait que la TI-Nspire la gère mérite d'être signalé.

2. L'algorithme d'Euclide étendu

2.1 *Identité de Bézout*

L'identité de Bézout précise que le pgcd de deux entiers peut s'écrire comme combinaison linéaire à coefficients entiers de ces deux entiers. Nous en rappelons la démonstration, en conservant les notations introduites précédemment (paragraphe 1.1).

Identité de Bézout

> *Chacun des restes* r_i peut s'écrire sous la forme $au_i + bv_i$, avec u_i et v_i entiers relatifs.
> **Conséquence immédiate :** c'est aussi le cas du pgcd, r_n…
> Il existe donc des coefficients u_n et v_n tels que $\text{pgcd}(a,b) = au_n + bv_n$.

Démonstration

Démontrons-le par récurrence. C'est vrai pour les deux premiers restes r_0 et r_1.

$r_0 = a - bq_0$: il suffit de prendre $u_0 = 1$ et $v_0 = -q_0$;
$r_1 = b - r_0q_1 = b - (a - bq_0)q_1 = (-q_1)a + (1 + q_0q_1)b$: il suffit cette fois de prendre $u_1 = -q_1$ et $v_1 = 1 + q_0q_1$...

Supposons que cela soit vrai pour deux restes r_{k-1} et r_k où k est un entier naturel arbitraire ; on a donc, pour k supérieur ou égal à 1 :

$$r_{k-1} = au_{k-1} + bv_{k-1} \text{ et } r_k = au_k + bv_k.$$

Montrons que cela est vrai pour le reste r_{k+1}. On peut écrire :

$$r_{k+1} = r_{k-1} - r_kq_{k+1} = (au_{k-1} + bv_{k-1}) - (au_k + bv_k)q_{k+1} = a(u_{k-1} - q_{k+1}u_k) + b(v_{k-1} - q_{k+1}v_k)$$

ce qui montre que r_{k+1} est bien de la forme attendue. En plus on a mis en évidence une relation de récurrence :

$$\begin{cases} u_{k+1} = u_{k-1} - q_{k+1} \times u_k \\ v_{k+1} = v_{k-1} - q_{k+1} \times v_k \end{cases}$$

Par conséquent, le pgcd, qui est un des restes obtenus (r_n), s'écrira bien sous la forme

$$au_n + bv_n$$

Remarquons que les trois suites *r*, *u* et *v* vérifient la même relation de récurrence, du type :

$$X_{n+1} = X_{n-1} - q_{k+1}X_n$$

où q_{k+1} est le quotient de r_{k-1} par r_k. Il est bien clair que les premiers termes sont différents (sinon les suites seraient égales).

Exemple : prenons a = 2 892 et b = 768.

Il est pratique de conserver les lettres, ici a et b, ce qui nous préservera de l'envie d'effectuer des calculs avec ces nombres. Notre seule préoccupation est d'exprimer successivement chacun des restes en fonction de a et b.

[2] Le programme *testpgcd* précédent, avec **pgcdr**, tourne pendant environ 3 minutes, contre 2 min 5 s pour **pgcd**.

$2\,892 = 768 \times 3 + 588 \quad \rightarrow \quad 588 = 2\,892 - 768 \times 3 = a - 3b$

$768 = 588 \times 1 + 180 \quad \rightarrow \quad 180 = 768 - 588 \times 1 = b - (a - 3b) = = -a + 4b$

$588 = 180 \times 3 + 48 \quad \rightarrow \quad 48 = 588 - 180 \times 3 = (a - 3b) - 3(-a + 4b)$
$= 4a - 15b$

$180 = 48 \times 3 + 36 \quad \rightarrow \quad 36 = 180 - 48 \times 3 = (-a + 4b) - 3(4a - 15b)$
$= -13a + 49b$

$48 = 36 \times 1 + 12 \quad \rightarrow \quad 12 = 48 - 36 = (4a - 15b) - (-13a + 49b)$
$= 17a - 64b$

$36 = 12 \times 3 + 0$ L'algorithme s'arrête…

Le pgcd est 12 et on a l'égalité : $12 = 17a - 64b = 17 \times 2\,892 - 64 \times 768$

2.2 *La description d'un premier algorithme*

On a vu plus haut que les trois suites *r*, *u* et *v* vérifient la *même* relation de récurrence :

$$\begin{cases} u_{i+1} = u_{i-1} - q_{i+1}u_i \\ v_{i+1} = v_{i-1} - q_{i+1}v_i \\ r_{i+1} = r_{i-1} - q_{i+1}r_i \end{cases}$$

Il faudrait donc, tout au long du calcul, conserver les deux plus récentes valeurs de ces suites, d'indice $i - 1$ et i, pour calculer celle qui vient après, d'indice $i+1$.

Se pose le problème des valeurs initiales. Observons ce qui se passe au départ :

de $a = bq_0 + r_0$, on tire $r_0 = a - bq_0$ donc on peut prendre $u_0 = 1$ et $v_0 = -q_0$;

de $b = r_0q_1 + r_1$, on tire $r_1 = b - r_0q_1 = b - (a - bq_0)q_1 = -q_1 \times a + (1 + q_0q_1) \times b$ donc on peut prendre $u_1 = -q_1$, et $v_1 = 1 + q_0q_1$.

Avec ces valeurs initiales, un problème de programmation apparaît. À cause de la présence de q_0 et de q_1 dans le calcul, ces valeurs initiales ne sont obtenues qu'à la fin des deux premières étapes de l'algorithme d'Euclide, et la relation de récurrence qui permet le calcul de ces suites ne peut être mise en œuvre qu'à partir de l'étape 3. Mais l'algorithme d'Euclide s'arrête parfois à la première ou deuxième étape… envisager ces cas particuliers risque fort de compliquer l'écriture de notre fonction…

Sauf si on a l'idée de changer d'uniformiser les notations en posant $a = r_{-2}$ et $b = r_{-1}$. Ce qui peut paraître surprenant de prime abord est en réalité très naturel. Ainsi :

$\boldsymbol{a} = bq_0 + r_0$

$\boldsymbol{b} = r_0q_1 + r_1$

$r_0 = r_1q_2 + r_2$

devient :

$\boldsymbol{r_{-2}} = r_{-1}q_0 + r_0$

$\boldsymbol{r_{-1}} = r_0q_1 + r_1$

$r_0 = r_1q_2 + r_2$

etc.

On remarque que la propriété sur les restes demeure valable, y compris pour r_{-2} et r_{-1}.

Comme

$$r_{-1} = b = 0 \times a + 1 \times b$$
$$= u_{-1} \times a + v_{-1} \times b,$$

en comparant les deux égalités, on peut poser :

$$\boldsymbol{u_{-1} = 0 \text{ et } v_{-1} = 1.}$$

De même, à partir de :

$$r_{-2} = a = 1 \times a + 0 \times b$$
$$= u_{-2} \times a + v_{-2} \times b,$$

on tire

$$\boldsymbol{u_{-2} = 1 \text{ et } v_{-2} = 0.}$$

Il est immédiat de vérifier qu'avec ces « nouvelles » valeurs initiales, on retrouve, avec la relation de récurrence, les premiers termes définis plus haut u_0 et u_1, ainsi que v_0 et v_1, des suites (u_i) et (v_i). On a ainsi :

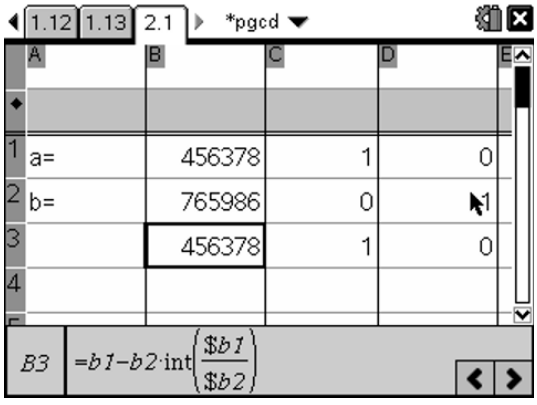
$$u_0 = u_{-2} - q_0 \times u_{-1} = 1 \text{ et } u_1 = u_{-1} - q_1 \times u_0 = -q_1,$$
$$v_0 = v_{-2} - q_0 \times v_{-1} = -q_0 \text{ et } v_1 = v_{-1} - q_1 \times v_0 = 1 + q_0 q_1.$$

2.3 *Algorithme d'Euclide étendu sur le tableur*

On dispose maintenant de tous les éléments pour concevoir notre feuille de calcul… Les colonnes B, C et D vont respectivement recevoir les valeurs des suites *r*, *u* et *v*. Procédons par étapes.

Tout d'abord, on rentre dans les deux premières lignes les valeurs de *a* et *b* (qui sont aussi les valeurs initiale r_{-2} et r_{-1} de la suite *r*), puis les valeurs initiales des suites *u* et *v* comme nous les avons décrites précédemment.

Ensuite, on s'occupe de la formule à mettre dans la cellule B3 : facile à retrouver en fait puisque c'est celle qui donne le premier reste, soit $r = a - bq$ donc **=b1–b2·int(b1/b2)**.



Des dollars ont été saisis pour permettre la recopie à droite (**CTRL C** dans B3 puis **CTRL V** dans C3 et D3). Si les cellules B1 et B2 de la formule doivent se transformer en C1 et C2 dans la formule de la colonne C, puis D1 et D2 dans la formule de la colonne D, le quotient, lui, par contre doit toujours être calculé de la même façon : il est impératif de fixer juste la colonne, en la précédant d'un $; la ligne quant à elle doit pouvoir changer quand on la recopiera vers le bas.

Il reste à recopier tout ceci vers le bas sur une dizaine de lignes : le pgcd est bien 26, avec les coefficients de Bézout à côté, mais les lignes qui suivent ne sont pas très belles…

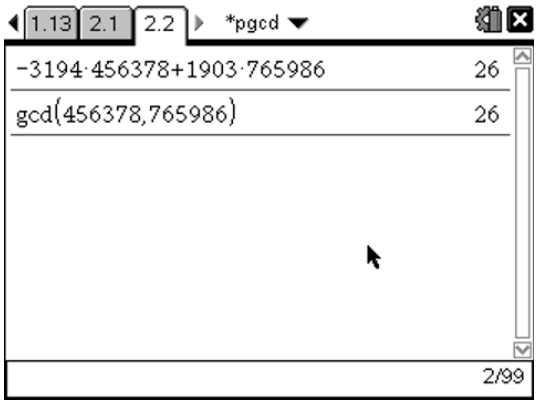
10		26	-3194	1903
11		0	29461	-17553
12		#UNDEF	-29461*...	17553*fl...

Il vaut donc mieux ne mener les calculs que lorsque le diviseur n'est pas nul : un **when** et des **void** règlent comme d'habitude tous nos problèmes, comme le montre la feuille de calcul qui suit :

	A	B	C	D	E	F	G	H	I	J	K	L
1	a=	456378	1	0								
2	b=	765986	0	1								
3		456378	1	0								
4		309608	-1	1								
5		146770	2	-1								
6		16068	-5	3								
7		2158	47	-28								
8		962	-334	199								
9		234	715	-426								
10		26	-3194	1903								
11		0	29461	-17553								
12		_	_	_								
13		_	_	_								
14		_	_	_								
15		_	_	_								
16		_	_	_								
17		_	_	_								

B3 =when($b2≠0, b1−b2·int($b1/$b2), _, _)

Évidemment, on peut vérifier que le pgcd de 456 378 et de 765 986 est 26 et on peut écrire la relation de Bézout suivante :

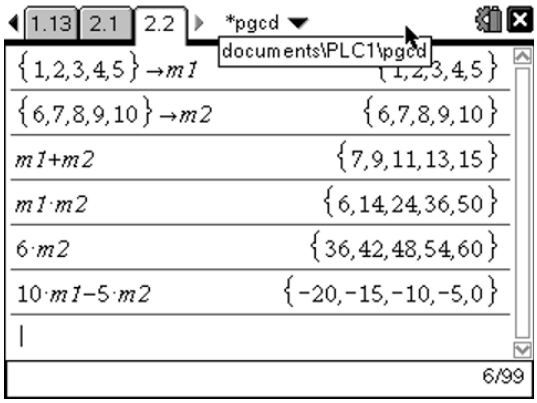


2.4 *Une fonction pour l'algorithme d'Euclide étendu*

Si le tableur présente un intérêt pédagogique certain, il reste limité dans le nombre de lignes qu'il traite à 2500. Pour un usage commode, l'algorithme doit pouvoir maintenant être traduit en TI-Basic.

La même relation de récurrence sert à trois suites : c'est un point qui va aussi nous servir. Même relation de récurrence, donc trois fois le même calcul… ou alors on utilise des listes…

L'avantage avec une liste réside dans la possibilité d'effectuer les calculs globalement, comme le montrent les exemples ci-après. Les calculs sont effectués terme à terme, pourvu que les deux listes aient effectivement le même nombre de termes…



Plutôt qu'effectuer trois calculs séparés, on n'en fera qu'un seul, portant sur une liste. Il nous faut donc stocker dans une liste les valeurs $\{r_i\ ;\ u_i\ ;\ v_i\}$ c'est-à-dire le reste r_i et les coefficients de la décomposition $r_i = u_i a + v_i b$.

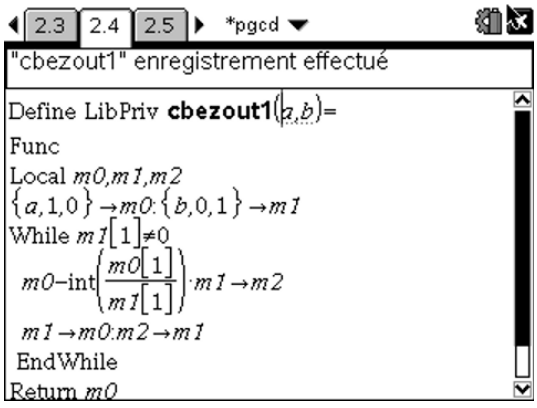
Deux étapes doivent être mémorisées pour calculer la suivante : la variable **m0** avec la liste $\{r_{i-1}\ ;\ u_{i-1}\ ;\ v_{i-1}\}$, la variable **m1** avec la liste $\{r_i\ ;\ u_i\ ;\ v_i\}$, la variable **m2** avec la liste $\{r_{i+1}\ ;\ u_{i+1}\ ;\ v_{i+1}\}$

Le quotient se calcule par $q_{i+1} = \text{int}\left(\frac{r_{i-1}}{r_i}\right) = \text{int}\left(\frac{m0[1]}{m1[1]}\right)$, stocké dans la variable q.

L'instruction **m0−q·m1→m2** applique, en un seul calcul portant sur une liste, la même relation de récurrence aux trois suites. Les deux listes **m0** et **m1** sont initialisées au début de la fonction comme on l'a indiqué plus haut.

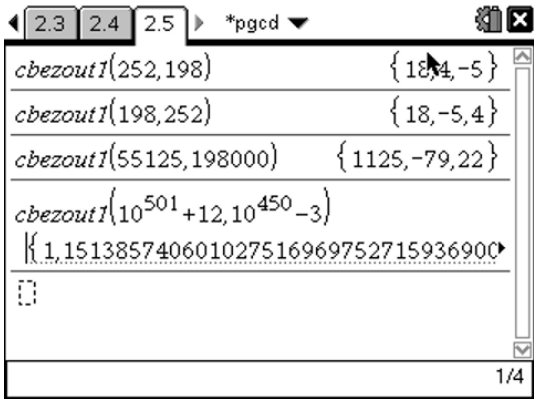
La fonction elle-même est gérée par une boucle **While** : le calcul simultané des termes de chacune des trois suites se fait en une seule instruction de liste ; puis on prépare les valeurs pour un éventuel autre passage dans la boucle. On sort de cette boucle dès que le reste est nul c'est-à-dire dès que **m1[1] = 0**: à ce moment, c'est la liste **m0** qui doit être renvoyée.

Le programme est donc le suivant :



Les résultats sont obtenus rapidement[3] comme d'habitude lorsque l'algorithme d'Euclide est en jeu, même quand les entiers sont de très grande taille :

[3] 2 min 22 s pour 30 000 utilisations dans un programme **test** analogue à celui du précédent paragraphe.



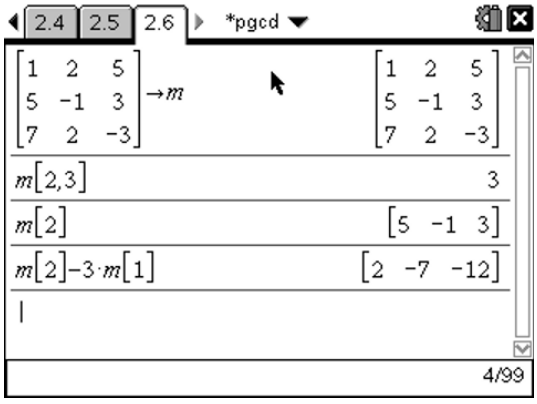
2.5 *Un pas de plus avec les matrices…*

On peut aussi gérer les calculs globalement par l'utilisation des matrices.

Supposons que l'on stocke les résultats dans une matrice du type :

$$m = \begin{pmatrix} r_{i-1} & u_{i-1} & v_{i-1} \\ r_i & u_i & v_i \\ r_{i+1} & u_{i+1} & v_{i+1} \end{pmatrix}$$

Ci-dessous quelques exemples d'utilisation des matrices avec TI-Nspire :



Quelques remarques sur l'utilisation des matrices avec TI-Nspire :

à la saisie, on sépare les lignes par des point-virgules, **[1,2,5;5,−1,3;7,2,−3]** dans notre cas ;

m[2,3] donne l'élément situé à l'intersection de la 2e *ligne* et de la troisième *colonne* (ici 3) ;

m[2] donne par contre la 2e ligne (ici **[5,−1,3]**) ;

enfin **m[2]−3·m[1]** effectue le calcul annoncé pour chacun des trois éléments de la ligne **m[2]** avec les éléments correspondant de la ligne **m[1].**

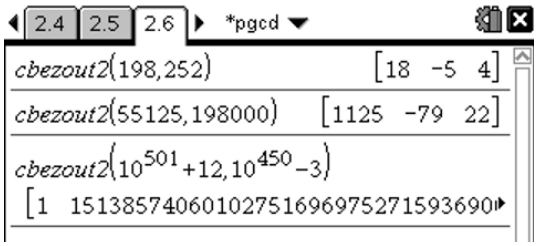
Comme pour les listes, une seule instruction permet le calcul de la troisième ligne à partir des deux premières :

m[1] – *q*·m[2]▸m[3] où q est le quotient de r_{i-1} par r_i.

Le programme y gagnera encore en simplicité. La matrice est initialisée à 0 par l'instruction **newMat** ; les deux premières lignes sont ensuite modifiées conformément aux valeurs initiales des suites.

```
cbezout2                                                    3/8
Define LibPriv cbezout2(a,b)=
Func
Local m
newMat(3,3)→m
[a 1 0]→m[1]:[b 0 1]→m[2]
While m[2,1]≠0
  m[1]-int(m[1,1]/m[2,1])·m[2]→m[3]
  m[2]→m[1]:m[3]→m[2]
 EndWhile
Return m[1]
EndFunc
```

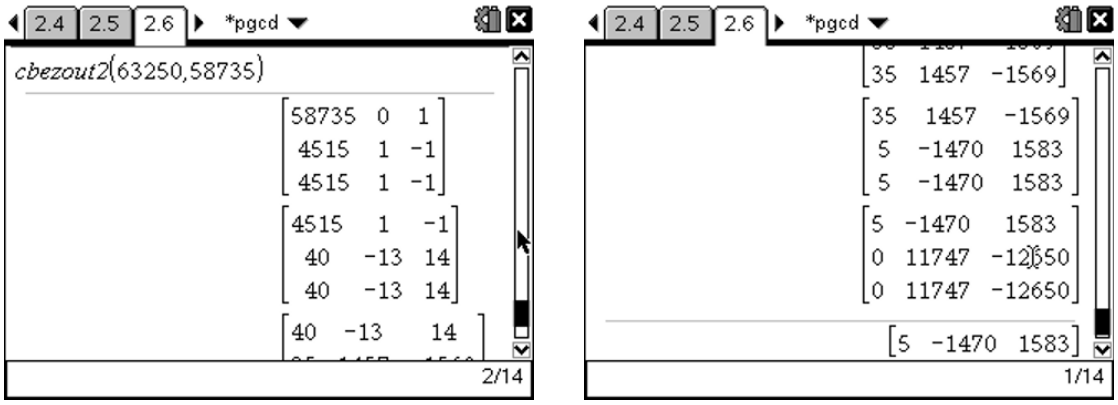
Pas de réel gain de temps au demeurant avec les résultats suivants[4] :



2.6 *L'algorithme de Blankinship*

Quand on examine les matrices traitées dans le programme précédent, on constate qu'il y a beaucoup de répétitions inutiles qui encombrent la mémoire et alourdissent les temps de calcul, au moins pour les très grand nombres.

Par exemple dans le calcul de **cbezout2(63250,58735)**, le programme gère successivement[5] :



On constate que deux lignes de la matrice sur les trois qu'elle contient sont strictement identiques. Et si on ne gardait que ce qui est utile ? Cette idée nous conduit à un algorithme décrit par le mathématicien Blankinship dans un article publié en 1963.

[4] Le même programme test que pour **cbezout1** demande cette fois 2 minutes 45 secondes…

[5] Ce que l'on obtient en demandant à chaque passage dans la boucle l'affichage, par **disp m,** de la matrice... C'est une instruction que l'on peut utiliser dans une fonction, sur une TI-Nspire : elle permet de suivre l'évolution d'une variable au cours du déroulement de la fonction.

À partir des matrices obtenues dans l'exemple ci-dessus, examinons comment l'on peut supprimer une ligne[6].

Le point de départ serait la matrice $M_0 = \begin{pmatrix} 63\,250 & 1 & 0 \\ 58\,735 & 0 & 1 \end{pmatrix}$.

À la fin de la première étape, on travaillerait avec la matrice $\begin{pmatrix} 58\,735 & 0 & 1 \\ 4\,515 & 1 & -1 \end{pmatrix}$ ou mieux avec $M_1 = \begin{pmatrix} 4\,515 & 1 & -1 \\ 58\,735 & 0 & 1 \end{pmatrix}$: puisqu'on travaille avec deux lignes, il est inutile de faire remonter la deuxième ligne, comme on le faisait avant.

Quelle transformation avons-nous appliquée sur cette matrice ? On a en fait remplacé la ligne L1 par la ligne $\mathrm{L1} - \mathrm{int}\left(\frac{63\,250}{58\,735}\right) \times \mathrm{L2} = \mathrm{L1} - \mathrm{L2}$, tandis que la ligne L2 est conservée. Cela ressemble à la méthode de Gauss de résolution des systèmes.

Étape suivante : il faut cette fois-ci se baser sur la deuxième ligne – c'est celle qui possède le plus fort pivot, si l'on cherche à poursuivre l'analogie avec la méthode de Gauss.

Comme $\mathrm{int}\left(\frac{58\,735}{4\,515}\right) = 13$, on remplace la ligne L2 par L2 – 13×L1 pour obtenir :

$M_2 = \begin{pmatrix} 4\,515 & 1 & -1 \\ 40 & -13 & 14 \end{pmatrix}$ … c'est bien une partie de la deuxième matrice rencontrée…

Et l'on poursuit suivant la même technique pour obtenir successivement :

en remplaçant la ligne L1 par L1 – 112×L2, $M_3 = \begin{pmatrix} 35 & 1\,457 & -1\,569 \\ 40 & -13 & 14 \end{pmatrix}$;

en remplaçant la ligne L2 par L2 – L1, $M_3 = \begin{pmatrix} 35 & 1\,457 & -1\,569 \\ 5 & -1\,470 & 1\,583 \end{pmatrix}$;

en remplaçant la ligne L1 par L1 – 7L2, $M_4 = \begin{pmatrix} 0 & 11\,747 & -12\,650 \\ 5 & -1\,470 & 1\,583 \end{pmatrix}$;

Le pgcd est 5 (… dernier reste non nul…), et les coefficients de la décomposition sont –1 470 et 1 583.

Théorème : l'algorithme de Blankinship

Soient *a* et *b* deux entiers naturels.
On fabrique la matrice $M = \begin{pmatrix} a & 1 & 0 \\ b & 0 & 1 \end{pmatrix}$.
On applique l'algorithme d'Euclide sur la première colonne, en partant toujours du plus grand nombre, qu'on appelle le plus grand « pivot ».
On étend les opérations faites sur la première colonne à toutes les lignes de la matrice *M*.

Une justification strictement matricielle peut être donnée, en complément du lien que nous venons de faire avec l'algorithme d'Euclide étendu.

[6] Le gain de place mémoire n'est pas négligeable, notamment si l'on travaille sur des grands nombres.

Soit $M=\begin{pmatrix}\alpha & \beta & \gamma\\ \delta & \varepsilon & \phi\end{pmatrix}$ une matrice à coefficients réels.

Remplacer la ligne L1 de cette matrice *M* par la ligne L1 + λL2 revient à multiplier cette matrice par la matrice $E_\lambda=\begin{pmatrix}1 & \lambda\\ 0 & 1\end{pmatrix}$: la vérification est immédiate.

De même, remplacer la ligne L2 de cette matrice par la ligne L2 + λL1 revient à multiplier cette matrice par la matrice $F_\lambda=\begin{pmatrix}1 & 0\\ \lambda & 1\end{pmatrix}$.

Lors de l'algorithme, on part de la matrice $\begin{pmatrix}a & 1 & 0\\ b & 0 & 1\end{pmatrix}$ pour arriver à la matrice $\begin{pmatrix}d & u & v\\ 0 & u' & v'\end{pmatrix}$, où *d* est le pgcd de *a* et de *b*, en appliquant des transformations du type précédent. On peut écrire :

$$\begin{pmatrix}d & u & v\\ 0 & u' & v'\end{pmatrix}=E'\begin{pmatrix}a & 1 & 0\\ b & 0 & 1\end{pmatrix}$$

où E' est une matrice produit de matrices du type précédent, soit E_λ ou F_λ.

En particulier, on peut en déduire : $\begin{pmatrix}d\\ 0\end{pmatrix}=E'\begin{pmatrix}a\\ b\end{pmatrix}$ et $\begin{pmatrix}u & v\\ u' & v'\end{pmatrix}=E'\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}=E'$.

Autrement dit, on récupère précisément la matrice *E'* dans les colonnes 2 et 3 du résultat… et comme $E'\begin{pmatrix}a\\ b\end{pmatrix}=\begin{pmatrix}d\\ 0\end{pmatrix}$ soit $\begin{pmatrix}u & v\\ u' & v'\end{pmatrix}\begin{pmatrix}a\\ b\end{pmatrix}=\begin{pmatrix}d\\ 0\end{pmatrix}$, on en déduit l'égalité *au* + *bv* = *d* que l'on cherchait.

Ci-dessous la fonction que l'on peut en déduire :

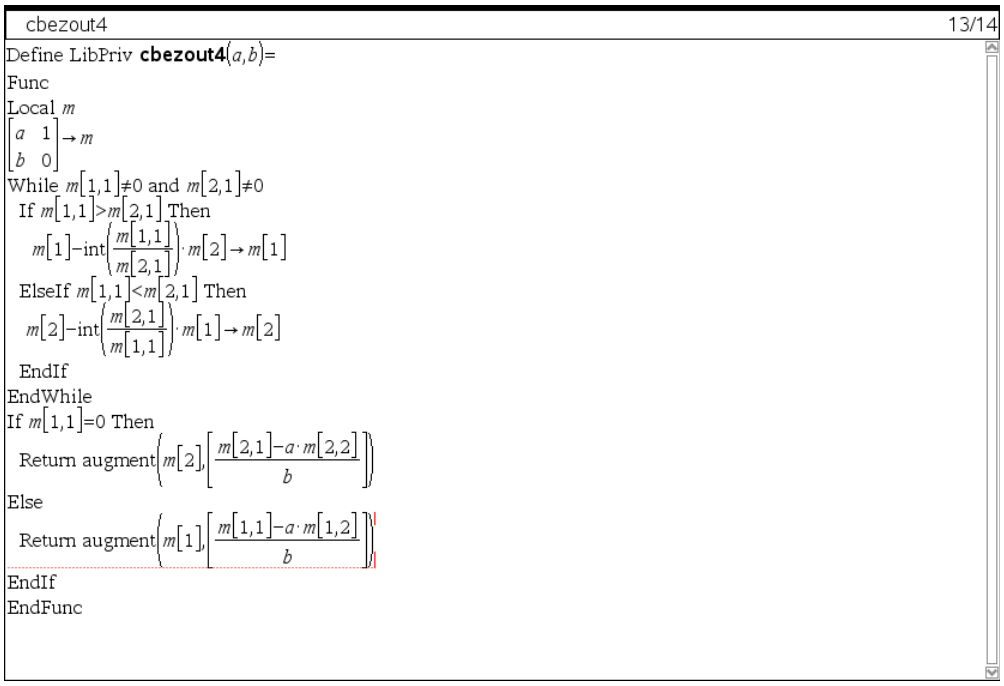
```
cbezout3                                                    7/14
Define LibPriv cbezout3(a,b)=
Func
Local m
[a 1 0]→m
[b 0 1]
While m[1,1]≠0 and m[2,1]≠0
 If m[1,1]>m[2,1] Then
  m[1]-int(m[1,1]/m[2,1])·m[2]→m[1]
 ElseIf m[1,1]<m[2,1] Then
  m[2]-int(m[2,1]/m[1,1])·m[1]→m[2]
 EndIf
EndWhile
If m[1,1]=0 Then
 Return m[2]
Else
 Return m[1]
EndIf
EndFunc
```

On obtient bien sûr les mêmes résultats qu'avec les fonctions précédentes[7].

[7] Léger gain en temps… 2 min 30 s pour le programme test…

Ultime affinage de l'algorithme de Blankinship

À bien y réfléchir, on n'a besoin que des deux premières colonnes de la matrice... La connaissance de *d*, le pgcd, *u*, *a* et *b* permet simplement de récupérer *v*. Ceci allège un peu les matrices utilisées… au détriment sans doute de la clarté du code…



Le gain en temps, par rapport à la fonction précédente, est par contre très minime.

Annexe : efficacité de l'algorithme d'Euclide

- **La suite de Fibonacci**

Nous l'étudierons dans un chapitre ultérieur. Elle a été introduite en 1202 dans le célèbre *Liber Abaci*, de Léonard de Pise, à propos de multiplication de lapins. Elle est définie par :

$$\begin{cases} f_0 = 1 \\ f_1 = 1 \\ \text{pour tout entier } n \geq 1,\ f_{n+2} = f_{n+1} + f_n \end{cases}$$

Les premiers termes de cette suite peuvent être calculés dans une feuille de calcul :

	A	B	C	D	E	F	G	H	I	J
◆	=seq(k,k,0,20)									
1	0	1								
2	1	1								
3	2	2								
4	3	3								
5	4	5								
6	5	8								
7	6	13								
8	7	21								
9	8	34								
10	9	55								
11	10	89								
12	11	144								
13	12	233								
14	13	377								
15	14	610								
16	15	987								
17	16	1597								
18	17	2584								
	B1	1								

Nous aurons besoin du résultat suivant :

> Pour tout entier naturel *n*, alors $f_n \geqslant \varphi^{n-1}$ où $\varphi = \dfrac{1+\sqrt{5}}{2}$.

Le réel φ est le célèbre nombre d'or, solution de l'équation $x^2 = x+1$. En particulier, $\varphi^2 = \varphi + 1$.

Démonstration

Montrons cette propriété par récurrence.

Elle est clairement vraie pour $n = 0$ ($u_0 = 1 \geqslant 1/\varphi$) ou pour $n = 1$ ($u_1 = 1 \geqslant \varphi^0 = 1$).

Supposons la propriété vraie pour deux entiers arbitraires p et $p + 1$. On peut alors affirmer :

$$f_p \geq \varphi^{p-1} \quad \text{et} \quad f_{p+1} \geq \varphi^p$$

Par addition, il vient $f_p + f_{p+1} \geq \varphi^{p-1} + \varphi^p = \varphi^{p-1}(1+\varphi) = \varphi^{p-1}\varphi^2 = \varphi^{p+1}$ soit $f_{p+2} \geq \varphi^{p+1}$, ce qui prouve bien la propriété pour tout entier.

- **Le résultat de Lamé**

Gabriel Lamé[8], nommé académicien l'année précédente[9], publie sa note sur la performance de l'algorithme d'Euclide en 1844 dans les *Comptes-rendus hebdomadaires de l'Académie des sciences.* Il écrit :

> *Dans les traités d'Arithmétique, on se contente de dire que le nombre des divisions à effectuer, dans la recherche du plus grand commun diviseur entre deux entiers, ne pourra pas surpasser la moitié du plus petit. Cette limite, qui peut être dépassée si les nombres sont petits, s'éloigne outre mesure quand ils ont plusieurs chiffres.*

Il établit dans la même note le théorème suivant :

> *Le nombre de divisions à effectuer, pour trouver le plus grand commun diviseur entre deux entiers A, et B < A, est toujours moindre que cinq fois le nombre des chiffres de B.*

C'est le résultat que nous allons prouver. Reprenons les notations précédentes en supposant que $a > b$. La suite des divisions successives de l'algorithme d'Euclide s'écrit alors :

division de a par b : $a = bq_0 + r_0$ avec $0 \leqslant r_0 < b$

division de b par r_0 : $b = r_0q_1 + r_1$ avec $0 \leqslant r_1 < r_0$

division de r_0 par r_1 : $r_0 = r_1q_2 + r_2$ avec $0 \leqslant r_2 < r_1$

…

division de r_{i-1} par r_i : $r_{i-1} = r_iq_{i+1} + r_{i+1}$ avec $0 \leqslant r_{i+1} < r_i$

…

division de r_{n-2} par r_{n-1} : $r_{n-2} = r_{n-1}q_n + r_n$ avec $0 \leqslant r_n < r_{n-1}$

division de r_{n-1} par r_n : $r_{n-1} = r_nq_{n+1} + 0$

Le pgcd est r_n, dernier reste non nul : il est obtenu au bout de $n + 2$ divisions.

Il s'apppuie dans sa démonstration sur un lien qu'il établit entre l'algorithme d'Euclide et la suite de Fibonacci.

Montrons d'abord que $b \geqslant r_0 + r_1$ puis que $r_k \geqslant r_{k+1} + r_{k+2}$ pour tout entier naturel k.

Partons à l'envers de $r_0 + r_1$:

$r_0 + r_1 = r_0 + b - r_0q_1 = b - r_0(q_1 - 1) \leqslant b$ car $q_1 \geqslant 1$ puisque $r_0 < b$.

Par un raisonnement analogue :

$$r_{k+1} + r_{k+2} = r_{k+1} + r_k - r_{k+1}q_{k+2} = r_k - (q_{k+2} - 1)r_{k+1} \leq r_k$$

car $q_{k+2} \geqslant 1$ pour les mêmes raisons…

Mais par ailleurs, on sait que $r_n \geqslant 1 = f_1$ et que $r_{n-1} > r_n$ donc $r_{n-1} \geqslant f_2$. L'inégalité précédente nous permet de remonter dans l'algorithme d'Euclide.

$r_{n-2} \geqslant r_{n-1} + r_n \geqslant f_2 + f_1 = f_3$

$r_{n-3} \geqslant r_{n-2} + r_{n-1} \geqslant f_3 + f_2 = f_4$

...

$r_2 \geqslant r_3 + r_4 \geqslant f_{n-2} + f_{n-3} = f_{n-1}$

[8] Je m'appuie sur l'excellent article de la revue des IREM, Repères, écrit par Norbert Verdier, Olivier Bordelles, Bernard Schott et Jean-Jacques Seitz, disponible en ligne : http://www.univ-irem.fr/reperes/articles/73_article_498.pdf.

[9] Il a aussi travaillé quelques années auparavant à l'étude du tracé de la ligne de chemin de fer Paris-Le Pecq.

$r_1 \geqslant r_2 + r_3 \geqslant f_{n-1} + f_{n-2} = f_n$

$r_0 \geqslant r_1 + r_2 \geqslant f_n + f_{n-1} = f_{n+1}$

Et donc $\boldsymbol{b \geqslant r_0 + r_1 \geqslant f_n + f_{n+1} \geqslant f_{n+2}}$

L'information, si elle peut paraître anodine, ne l'est pas du tout : elle relie *b* au nombre total de divisions à faire, par l'intermédiaire de la suite de Fibonacci.

Supposons par exemple que *b* soit inférieur strictement à 10 000. On a nécessairement :

$f_{n+2} \leq 10\,000$ soit $n + 2 \leq 19$ car $f_{19} = 6\,765$ et $f_{20} = 10\,946$

Le nombre de divisions, soit $n + 2$, est majoré par 19… soit au maximum 5 fois le nombre de chiffres de *b* comme l'annonce Lamé.

Précisons ce résultat en supposant que *b* possède au plus *k* chiffres, soit que $b < 10^k$.

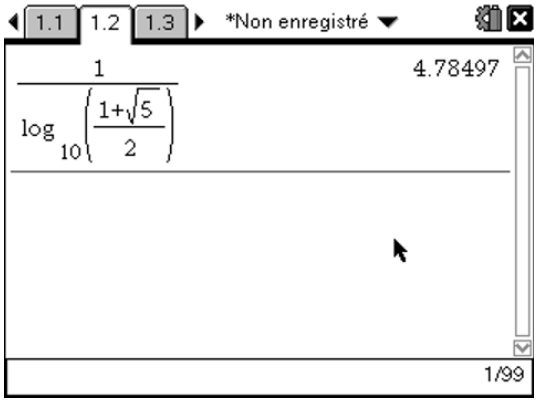
On peut donc écrire $f^{n+2} \leq b$ et en déduire :

$\varphi^{n+1} \leq f^{n+2} \leq b < 10^k$

En prenant le logarithme décimal des inégalités extrêmes, on en déduit :

$$(n+1)\log\varphi < k \text{ soit } n+1 < \frac{k}{\log\varphi} < 5k$$

d'après les calculs suivants :

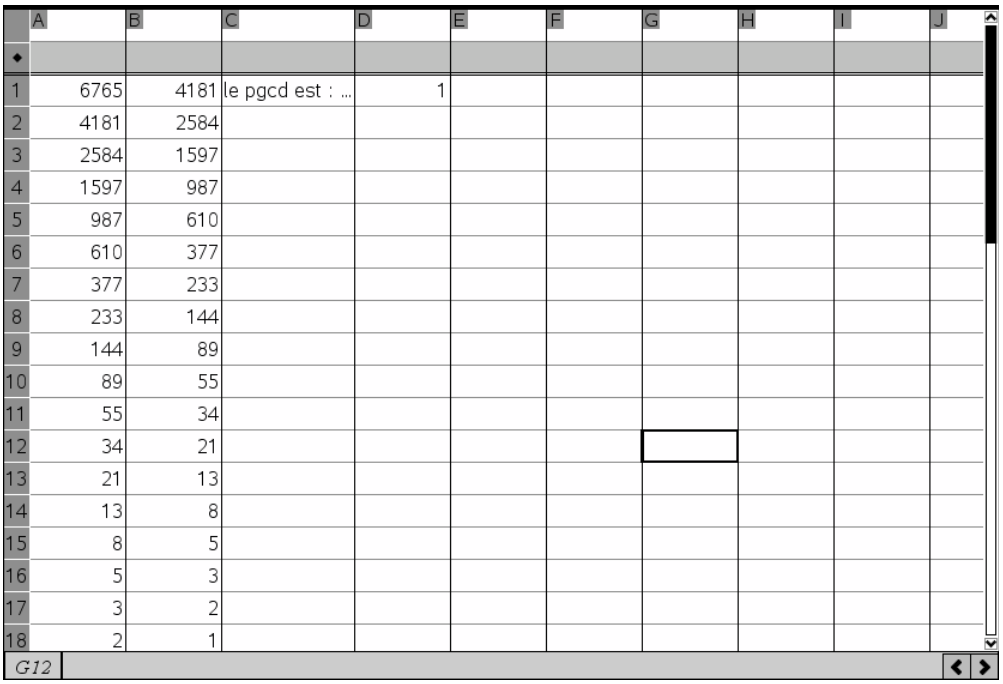


En particulier, on peut en déduire que le nombre de divisions, $n + 2$ est inférieur ou égal à 5 fois le nombre de chiffres de *b*, comme l'a indiqué Lamé.

Ceci montre la performance d'un tel algorithme : ainsi avec un nombre *b* possédant 20 chiffres, on aura au plus 100 divisions à effectuer et ce quelle que soit la taille de l'autre nombre *a*, pourvu qu'il soit plus grand que *b* !

Il est facile de voir que le calcul le pgcd de $f_{19} = 6\,765$ et $f_{18} = 4\,181$ demandera précisément 19 étapes : on redescend successivement toutes les valeurs de la suite de Fibonacci jusqu'aux indices 1 et 0.

On obtient là quasiment le maximum prévu par Lamé pour les nombres à 4 chiffres. La majoration obtenue est bien la plus petite possible.



	A	B	C	D	E	F	G	H	I	J
1	6765	4181	le pgcd est : ...	1						
2	4181	2584								
3	2584	1597								
4	1597	987								
5	987	610								
6	610	377								
7	377	233								
8	233	144								
9	144	89								
10	89	55								
11	55	34								
12	34	21								
13	21	13								
14	13	8								
15	8	5								
16	5	3								
17	3	2								
18	2	1								

G12