Algorithmes de-ci de-là

Journées académiques IREM/APMEP juin 2009

Guillaume CONNAN

Version longue disponible à cette adresse
http://download.tuxfamily.org/tehessinmath/les%20pdf/PafAlgo.pdf

10 juin 2009

Sommaire

Sommaire

Ce n'est pas très mathématique mais c'est un algorithme et pour affronter tous ceux qui nous attendent, il va nous falloir des forces....

Entrées : masse m totale
Initialisation : ;
beurre$\longleftarrow$ m/4;
sucre$\longleftarrow$ m/4;
farine$\longleftarrow$ m/4;
œuf $\longleftarrow$ m/4;
début
Couper le beurre en petits moreaux et le mettre à fondre doucement au bain-marie de préférence. Dès qu'il est fondu arrêter. Laisser refroidir mais attention : le beurre doit être encore liquide ! Il ne doit pas redevenir solide;
Mettre le four à préchauffer à 160°(th 5);
Mettre les oeufs entiers avec le sucre dans un saladier. Battre longuement le mélange pour qu'il blanchisse et devienne bien mousseux;
Y ajouter le beurre fondu ET FROID;
Rajouter progressivement à l'appareil la farine en l'incoporant bien. Cela doit donner une pâte élastique et un peu épaisse;
Verser la préparation dans un moule à manqué ou à cake bien beurré;
Laisser cuire environ une heure. Il faut surveiller le gâteau régulièrement.;
si *il semble brunir trop vite* alors
  il faut baisser un peu le four et mettre une feuille d'aluminium sur le dessus.
Il faut que le dessus du gâteau soit blond foncé, mais pas trop coloré.;
si *lorsqu'une pique plantée en son milieu ressort sèche* alors
  le gâteau est cuit
fin

Algorithme 1 : algorithme breton pur beurre

Sommaire

Sommaire

Les enfants apprennent à l'école primaire à poser « verticalement » une multiplication d'entier : ils utilisent un des nombreux algorithmes utilisés à travers les siècles et les continents pour effectuer ce type de calcul.

Celui-ci nous intéresse car c'est une première version d'un algorithme plus général sur le multiplication des polynômes et même des fractions rationnelles.

Les enfants apprennent à l'école primaire à poser « verticalement » une multiplication d'entier : ils utilisent un des nombreux algorithmes utilisés à travers les siècles et les continents pour effectuer ce type de calcul.
Celui-ci nous intéresse car c'est une première version d'un algorithme plus général sur le multiplication des polynômes et même des fractions rationnelles.

En effet, derrière l'algorithme vu en primaire se cache celui du produit des polynômes etfaire le lien entre ces deux applications de cet algorithme nécessite une bonne compréhension de la numération en base 10.

$$\begin{array}{r} \times \begin{array}{rrrr} 4 & 0 & 1 & 2 \\ & & 1 & 0 & 2 \end{array} \\ \hline \begin{array}{rrrrrr} & & 8 & 0 & 2 & 4 \end{array} \\ \begin{array}{rrrrrr} 4 & 0 & 1 & 2 & \cdot & \cdot \end{array} \\ \hline \begin{array}{rrrrrr} 4 & 0 & 9 & 2 & 2 & 4 \end{array} \end{array}$$

$$(4 \times 10^3 + 10 + 2) \times (10^2 + 2) = (8 \times 10^3 + 2 \times 10 + 4) + (4 \times 10^5 + 10^3 + 2 \times 10)$$

$$(4X^3 + X + 2)(X^2 + 2) = (8X^3 + 2X + 4) + (4X^5 + X^3 + 2 \times X)$$

et même

$$10 \times 31,7 = 317 \parallel X(3X + 1 + \frac{7}{X} = 3X^2 + X + 7)$$

$$\begin{array}{rr} & 4012 \\ \times & 102 \\ \hline & 8024 \\ 4012\cdot & \cdot \\ \hline 409224 & \end{array}$$

$$(4\times10^3+10+2)\times(10^2+2)=(8\times10^3+2\times10+4)+(4\times10^5+10^3+2\times10)$$

$$(4X^3+X+2)(X^2+2)=(8X^3+2X+4)+(4X^5+X^3+2\times X)$$

et même

$$10\times31,7=317 \parallel X(3X+1+\frac{7}{X}=3X^2+X+7)$$

$$\begin{array}{rr} & 4012 \\ \times & 102 \\ \hline & 8024 \\ 4012\cdot & \cdot \\ \hline 409224 & \end{array}$$

$$(4\times10^3+10+2)\times(10^2+2)=(8\times10^3+2\times10+4)+(4\times10^5+10^3+2\times10)$$

$$(4X^3+X+2)(X^2+2)=(8X^3+2X+4)+(4X^5+X^3+2\times X)$$

et même

$$10\times31,7=317 \parallel X(3X+1+\frac{7}{X}=3X^2+X+7)$$

$$\begin{array}{rr} & 4\,0\,1\,2 \\ \times & \quad 1\,0\,2 \\ \hline & 8\,0\,2\,4 \\ 4\,0\,1\,2 & \cdot\ \cdot \\ \hline 4\,0\,9\,2\,2\,4 & \end{array}$$

$$\left(4\times10^3+10+2\right)\times\left(10^2+2\right)=\left(8\times10^3+2\times10+4\right)+\left(4\times10^5+10^3+2\times10\right)$$

$$\left(4X^3+X+2\right)\left(X^2+2\right)=\left(8X^3+2X+4\right)+\left(4X^5+X^3+2\times X\right)$$

et même

$$10\times31,7=317 \parallel X(3X+1+\frac{7}{X}=3X^2+X+7)$$

Sommaire

```
4012*102
```

409224

```
f:=x->(4x^3+x+2)*(x^2+2):;
```

```
f(10)
```

409224

```
4012*102
```

409224

```
f:=x->(4x^3+x+2)*(x^2+2):;
```

```
f(10)
```

409224

```
4012*102
```

409224

```
f:=x->(4x^3+x+2)*(x^2+2):;
```

```
f(10)
```

409224

```
4012*102
```

409224

```
f:=x->(4x^3+x+2)*(x^2+2):;
```

```
f(10)
```

409224

```
4012*102
```

409224

```
f:=x->(4x^3+x+2)*(x^2+2):;
```

```
f(10)
```

409224

```
produit(N,M,var):={
local PN,PM,Pro,k;
PN:=0;PM:=0;
 for(k:=0;k<size(N);k++){
    PN:=PN+N[k]*X^(size(N)-1-k)
  }
 for(k:=0;k<size(M);k++){
    PM:=PM+M[k]*X^(size(M)-1-k)
  }
Pro:=X->PM*PN;
return(simplify(expand(Pro(var))))
}:;
```

```
produit([4,0,1,2],[1,0,2],10)
```

409224

```
produit([4,0,1,2],[1,0,2],x)
```

$4x^5 + 9x^3 + 2x^2 + 2x + 4$

```
produit([4,0,1,2],[1,0,2],10)
```

409224

```
produit([4,0,1,2],[1,0,2],x)
```

$4x^5 + 9x^3 + 2x^2 + 2x + 4$

```
produit([4,0,1,2],[1,0,2],10)
```

409224

```
produit([4,0,1,2],[1,0,2],x)
```

$4x^5 + 9x^3 + 2x^2 + 2x + 4$

```
produit([4,0,1,2],[1,0,2],10)
```

409224

```
produit([4,0,1,2],[1,0,2],x)
```

$4x^5 + 9x^3 + 2x^2 + 2x + 4$

Sommaire

La relation $u_{n+1} = 3u_n + 2$ valable pour tout entier naturel n et la donnée de $u_0 = 5$ constituent un algorithme de calcul récursif de tout terme de la suite (u_n).

Sa traduction dans des langages sachant plus ou moins traiter de tels algorithmes est directe.

La relation $u_{n+1} = 3u_n + 2$ valable pour tout entier naturel n et la donnée de $u_0 = 5$ constituent un algorithme de calcul récursif de tout terme de la suite (u_n).
Sa traduction dans des langages sachant plus ou moins traiter de tels algorithmes est directe.

Par exemple, en XCAS cela devient :

```
u(n,uo):={
if(n==0){return(uo)}
else{3*u(n-1,uo)+2}
}
```

et en CAML :

```
# let rec u(n,uo)=
    if n=0 then uo
    else 3*u(n-1,uo)+2;;
```

Par exemple, en XCAS cela devient :

```
u(n,uo):={
if(n==0){return(uo)}
else{3*u(n-1,uo)+2}
}
```

et en CAML :

```
# let rec u(n,uo)=
    if n=0 then uo
    else 3*u(n-1,uo)+2;;
```

Par exemple, en XCAS cela devient :

```
u(n,uo):={
if(n==0){return(uo)}
else{3*u(n-1,uo)+2}
}
```

et en CAML :

```
# let rec u(n,uo)=
    if n=0 then uo
    else 3*u(n-1,uo)+2;;
```

Par exemple, en XCAS cela devient :

```
u(n,uo):={
if(n==0){return(uo)}
else{3*u(n-1,uo)+2}
}
```

et en CAML :

```
# let rec u(n,uo)=
    if n=0 then uo
    else 3*u(n-1,uo)+2;;
```

Deux écoles de programmation :

- programmation fonctionnelle ;
- programmation impérative.

Deux écoles de programmation :

- programmation fonctionnelle ;
- programmation impérative.

Cela donne en XCAS :

```
u_imp(n,uo):={
local k,temp;
temp:=uo;
for(k:=1;k<=n;k:=k+1){
temp:=3*temp+2;
}
return(temp);
}
```

Cela donne en XCAS :

```
u_imp(n,uo):={
local k,temp;
temp:=uo;
for(k:=1;k<=n;k:=k+1){
temp:=3*temp+2;
}
return(temp);
}
```

Sommaire

Certains connaissent peut-être le langage Logo qui avait permis aux petits écoliers d'il y a une vingtaine d'années de s'initier à la programmation. Voyons par exemple un petit algorithme permettant de tracer un polygone fermé régulier quelconque.

Entrées : nombre N de côtés et longueur a d'un côté

début

  pour *k de 1 jusqu'à N* **faire**

    avance de a pas;

    tourne de 360/a degrés;

fin

Algorithme 2 : polygone

Avec XCAS

XCAS comprend un module *Tortue* très complet :

```
polylogo(N,a):={
local k;
pour k de 1 jusque N faire
avance(a);
tourne_droite(360/N);
fpour;
}:;
```

Avec XCAS

XCAS comprend un module *Tortue* très complet :

```
polylogo(N,a):={
local k;
pour k de 1 jusque N faire
avance(a);
tourne_droite(360/N);
fpour;
}:;
```

Sommaire

Sommaire

Algorithme

On peut définir la valeur absolue d'un réel x de la manière suivante, même si ce n'est pas l'usage en seconde :

```
Entrées : x(un réel)
début
    si x positif alors
        retourner x
    sinon
        retourner (-x)
fin
```

Algorithme 3 : valeur absolue

Algorithme

On peut définir la valeur absolue d'un réel x de la manière suivante, même si ce n'est pas l'usage en seconde :

Entrées : x(un réel)
début
  si x *positif* **alors**
    retourner x
  sinon
    retourner (-x)
fin

Algorithme 4 : valeur absolue

Avec XCAS

```
va(x):={
  si x>0 alors
    return(x) sinon
    return(-x)
  fsi;
}
```

en anglais :

```
va(x):={
   if(x>0){return(x)}
   else{return(-x)}
}
```

en anglais :

```
va(x):={
   if(x>0){return(x)}
   else{return(-x)}
}
```

en plus compact :

```
va(x):={
ifte(x>0,x,-x)
}
```

Avec CAML

```
# let va(x)=
    if x>0 then x else -x;;
```

Sommaire

Algorithme récursif

On part du fait que la partie entière d'un nombre appartenant à $[0;1[$ est nulle.

Ensuite, on « descend » de x vers 0 par pas de 1 si le nombre est positif en montant que $\lfloor x \rfloor = 1 + \lfloor x - 1 \rfloor$.

Si le nombre est négatif, on « monte » vers 0 en montrant que $\lfloor x \rfloor = -1 + \lfloor x + 1 \rfloor$.

Algorithme récursif

On part du fait que la partie entière d'un nombre appartenant à $[0;1[$ est nulle.
Ensuite, on « descend » de x vers 0 par pas de 1 si le nombre est positif en montant que $\lfloor x \rfloor = 1 + \lfloor x - 1 \rfloor$.
Si le nombre est négatif, on « monte » vers 0 en montrant que $\lfloor x \rfloor = -1 + \lfloor x + 1 \rfloor$.

Algorithme récursif

On part du fait que la partie entière d'un nombre appartenant à $[0;1[$ est nulle.
Ensuite, on « descend » de x vers 0 par pas de 1 si le nombre est positif en montant que $\lfloor x \rfloor = 1 + \lfloor x - 1 \rfloor$.
Si le nombre est négatif, on « monte » vers 0 en montrant que $\lfloor x \rfloor = -1 + \lfloor x + 1 \rfloor$.

Avec XCAS

```
partie_entiere_rec(r):={
  si(r>=0) et (r<1)
       alors 0
       sinon si r>0
             alors 1+partie_entiere_rec(r-1)
             sinon -1+partie_entiere_rec(r+1)
             fsi
       fsi
};;
```

```
partie_entiere_rec(r):={
  if((r>=0) and (r<1)){0}
  else{if(r>0){1+partie_entiere_rec(r-1)}
       else{-1+partie_entiere_rec(r+1)}
      }
};;
```

Avec XCAS

```
partie_entiere_rec(r):={
  si(r>=0) et (r<1)
       alors 0
       sinon si r>0
             alors 1+partie_entiere_rec(r-1)
             sinon -1+partie_entiere_rec(r+1)
             fsi
       fsi
};;
```

```
partie_entiere_rec(r):={
  if((r>=0) and (r<1)){0}
  else{if(r>0){1+partie_entiere_rec(r-1)}
       else{-1+partie_entiere_rec(r+1)}
      }
};;
```

Avec CAML

```
# let rec partie_entiere (r)
        =
        if r >= 0. && r < 1.
         then 0.
         else if r > 0.
                then   1. +. partie_entiere (r -.
                    1.)
       else -.1. +. partie_entiere (r +. 1.)
```

Algorithme impératif

Test « si...alors...sinon » et boucle « tant que »

On peut commencer par se restreindre aux nombres positifs. On part de 0. Tant qu'on n'a pas dépassé x, on avance d'un pas. La boucle s'arrête dès que k est strictement supérieur à x. La partie entière vaut donc $k-1$.

Algorithme impératif

Test « si...alors...sinon » et boucle « tant que »
On peut commencer par se restreindre aux nombres positifs. On part de 0. Tant qu'on n'a pas dépassé x, on avance d'un pas. La boucle s'arrête dès que k est strictement supérieur à x. La partie entière vaut donc $k - 1$.

Algorithme impératif

Entrées : x(réel positif)
Initialisation : k $\longleftarrow$ 0
début
  tant que $k<=x$ **faire**
    k$\longleftarrow$ k+1
  retourner *k-1*
fin

Algorithme 5 : partie entière d'un nombre positif

Algorithme impératif

Dans le cas d'un réel négatif, il faut cette fois reculer d'un pas à chaque tour de boucle et la partie entière est la première valeur de k à être inférieure à x :

Algorithme impératif

Entrées : x(réel)
Initialisation : k $\longleftarrow$ 0
début
  si $x > 0$ **alors**
    tant que $k<=x$ **faire**
      k$\longleftarrow$ k+1
    retourner *k-1*
  sinon
    tant que $k>x$ **faire**
      k$\longleftarrow$ k-1
    retourner k
fin

Algorithme 6 : partie entière d'un réel quelconque

Algorithme impératif avec XCAS

```
pe(x):={
local k;
k:=0;
si x>=0 alors
    tantque k<=x faire
        k:=k+1;
    ftantque;
    return(k-1);
sinon
    tantque k>x faire
        k:=k-1;
    ftantque;
    return(k);
fsi;
};;
```

En anglais :

```
pe(x):={
local k;
k:=0;
if(x>=0){
    while(k<=x){k:=k+1};
    return(k-1);
          }
else{
    while(k>x){k:=k-1};
    return(k);
     }
};;
```

Sommaire

On veut arrondir un nombre réel à 10^{-2} près par défaut.

On va prendre la partie entière (avec l'algorithme créé précédemment) du nombre multiplié par 10^2 puis le rediviser par 10^2 :

Entrées : x(réel)
début
  retourner *(partie entière de* $100x$*) divisée par 100*
fin

Algorithme 7 : arrondi

On veut arrondir un nombre réel à 10^{-2} près par défaut.
On va prendre la partie entière (avec l'algorithme créé précédemment) du nombre multiplié par 10^2 puis le rediviser par 10^2 :

Entrées : x(réel)
début
  retourner *(partie entière de* $100x$*) divisée par 100*
fin

Algorithme 8 : arrondi

On veut arrondir un nombre réel à 10^{-2} près par défaut.
On va prendre la partie entière (avec l'algorithme créé précédemment) du nombre multiplié par 10^2 puis le rediviser par 10^2 :

Entrées : x(réel)
début
  retourner *(partie entière de* $100x$*) divisée par 100*
fin

Algorithme 9 : arrondi

Traduction XCAS

```
arrondi(x):={
return(pe(100*x)/100.0)
}
```

Traduction CAML

```
let arrondi_au_100_eme_par_defaut (r)
    =
    partie_entiere (100. *. r) /. 100.
```

Sommaire

On veut par exemple calculer la somme des entiers de 1 à n.

Algorithme récursif

En XCAS :

```
som_ent(n):={
if(n==0){0}
else{n+som_ent(n-1)}
}:;
```

En CAML :

```
# let rec som_ent(n)=
    if n=0 then 0
    else n+som_ent(n-1);;
```

Algorithme récursif

En XCAS :

```
som_ent(n):={
if(n==0){0}
else{n+som_ent(n-1)}
}:;
```

En CAML :

```
# let rec som_ent(n)=
    if n=0 then 0
    else n+som_ent(n-1);;
```

Algorithme impératif

Entrées : n(entier naturel)
Initialisation : S$\longleftarrow$ 0; *// la somme est nulle au départ*
début
  pour *k de 1 à n* **faire**
    S$\longleftarrow$ S+k
  retourner *S*
fin

Algorithme 10 : somme des entiers de 1 à n

Traduction XCAS

```
som(n):={
local S;
S:=0;
pour k de 1 jusque n faire
  S:=S+k;
fpour;
return(S)
}
```

en anglais :

```
som(n):={
local S;
S:=0;
for(k:=1;k<=n;k++){S:=S+k};
return(S)
}
```

Traduction XCAS

```
som(n):={
local S;
S:=0;
pour k de 1 jusque n faire
  S:=S+k;
fpour;
return(S)
}
```

en anglais :

```
som(n):={
local S;
S:=0;
for(k:=1;k<=n;k++){S:=S+k};
return(S)
}
```

Traduction XCAS

```
som(n):={
local S;
S:=0;
pour k de 1 jusque n faire
  S:=S+k;
fpour;
return(S)
}
```

en anglais :

```
som(n):={
local S;
S:=0;
for(k:=1;k<=n;k++){S:=S+k};
return(S)
}
```

Sommaire

Les sections STG sont initiées à l'algorithmique en travaillant sur des problèmes simples de gestion.

Sommaire

On entre un montant HT ht. On effectue une remise sur ht selon la régle suivante :

- si $ht < 2500$€ alors il n'y a pas de remise ;
- si $2500 \leqslant ht < 4000$€ alors la remise est de 5% ;
- dans les autres cas la remise est de 8%.

La TVA est de 19,6%. On demande le prix TTC en fonction du prix HT.

On entre un montant HT ht. On effectue une remise sur ht selon la régle suivante :

- si $ht < 2500$€ alors il n'y a pas de remise ;
- si $2500 \leqslant ht < 4000$€ alors la remise est de 5% ;
- dans les autres cas la remise est de 8%.

La TVA est de 19,6%. On demande le prix TTC en fonction du prix HT.

On entre un montant HT ht. On effectue une remise sur ht selon la régle suivante :

- si $ht < 2500$€ alors il n'y a pas de remise ;
- si $2500 \leqslant ht < 4000$€ alors la remise est de 5% ;
- dans les autres cas la remise est de 8%.

La TVA est de 19,6%. On demande le prix TTC en fonction du prix HT.

Algorithme

Entrées : HT(réel positif)
Initialisation : ht$\longleftarrow$ HT
début
  si *ht>=2500 et ht<4000* **alors**
    ht$\longleftarrow$ ht$\times$ 0,95
  si *ht>=4000* **alors**
    ht$\longleftarrow$ ht$\times$0,92
  retourner *ht*$\times$1.196
fin

Algorithme 11 : prix TTC selon remise

XCAS

```
remise(HT):={
local ht;
ht:=HT;
si (ht>=2500) et (ht<4000) alors ht:=ht*0.95; fsi;
si ht>=4000 alors ht:=ht*0.92; fsi;
return(ht*1.196)
 }
```

```
remise(HT):={
local ht;
ht:=HT;
 if( (ht>=2500) and (ht<4000)){ht:=ht*0.95};
 if(ht>=4000){ht:=ht*0.92};
return(ht*1.196)
 }
```

XCAS

```
remise(HT):={
local ht;
ht:=HT;
si (ht>=2500) et (ht<4000) alors ht:=ht*0.95; fsi;
si ht>=4000 alors ht:=ht*0.92; fsi;
return(ht*1.196)
 }
```

```
remise(HT):={
local ht;
ht:=HT;
 if( (ht>=2500) and (ht<4000)){ht:=ht*0.95};
 if(ht>=4000){ht:=ht*0.92};
return(ht*1.196)
 }
```

CAML

```
let prix_ttc_apres_remise (prix_hors_taxe)
    =
    let
     taux_remise = if prix_hors_taxe < 2500.
                   then 0.
                   else if prix_hors_taxe < 4000.
                        then 0.05
                        else 0.08
    in
     (1. -. taux_remise) *. prix_hors_taxe *.
        1.196
```

On remarque au passage comment définir une fonction locale imbriquée dans une autre fonction.

CAML

```
let prix_ttc_apres_remise (prix_hors_taxe)
    =
    let
     taux_remise = if prix_hors_taxe < 2500.
                   then 0.
                   else if prix_hors_taxe < 4000.
                        then 0.05
                        else 0.08
    in
     (1. -. taux_remise) *. prix_hors_taxe *.
        1.196
```

On remarque au passage comment définir une fonction locale imbriquée dans une autre fonction.

Sommaire

Sommaire

Version récursive

```
reste(a,b) début
    si a<b alors
        retourner 0
fin
retourner 1+quotient(a-b,b)
```

Algorithme 12 : Quotient : version récursive

Avec XCAS

```
quotient(a,b):={
if(a<b){return(0)};
return(1+quotient(a-b,b));
}
```

Avec CAML

```
# let rec quotient (a,b)
        =
        if a<b
         then 0
         else  1+quotient(a-b,b);;
```

Remarque

CAML étant intelligemment typé, si on entre :

```
# 37/5;;
```

On obtient

```
- : int = 7
```

c'est-à-dire par défaut le quotient entier car on travaille avec des entiers.

Pour avoir une approximation décimale, il faut entrer :

```
# 37./.5.;;
```

et on obtient :

```
- : float = 7.4
```

Pour le reste, on peut donc entrer :

```
# 37-37/5*5;;
```

ou utiliser la commande infixée `mod`

```
# 37 mod 2 ;;
```

Remarque

CAML étant intelligemment typé, si on entre :

```
# 37/5;;
```

On obtient

```
- : int = 7
```

c'est-à-dire par défaut le quotient entier car on travaille avec des entiers.

Pour avoir une approximation décimale, il faut entrer :

```
# 37./.5.;;
```

et on obtient :

```
- : float = 7.4
```

Pour le reste, on peut donc entrer :

```
# 37-37/5*5;;
```

ou utiliser la commande infixée `mod`

```
# 37 mod 2 ;;
```

Remarque

CAML étant intelligemment typé, si on entre :

```
# 37/5;;
```

On obtient

```
- : int = 7
```

c'est-à-dire par défaut le quotient entier car on travaille avec des entiers.

Pour avoir une approximation décimale, il faut entrer :

```
# 37./.5.;;
```

et on obtient :

```
- : float = 7.4
```

Pour le reste, on peut donc entrer :

```
# 37-37/5*5;;
```

ou utiliser la commande infixée `mod`

```
# 37 mod 2 ;;
```

Version impérative

Entrées : 2 entiers a et b

début

  Initialisation : k$\longleftarrow$ 0

  tant que $k \times b \leqslant a$ **faire**

    k$\longleftarrow$ k+1

fin

retourner *k-1*

Algorithme 13 : Quotient : version impérative

Avec XCAS

```
quotient(a,b):={
local k;
k:=0;
while(k*b<=a){k:=k+1}
return(k-1)
}
```

Sommaire

On peut créer une fonction qui testera si un nombre divise un autre.

Entrées : 2 entiers n et p

début

| (quotient entier n/p)× p == n

fin

Algorithme 14 : Test de divisibilité

On effectue un test booléen en utilisant l'opérateur infixe == qui désigne la relation d'égalité. La réponse du programme sera donc « vrai » ou « faux » (« 1 » ou « 0 » sur XCAS et « true » ou « false » sur CAML)

On peut créer une fonction qui testera si un nombre divise un autre.

Entrées : 2 entiers n et p

début

| (quotient entier n/p)× p == n

fin

Algorithme 15 : Test de divisibilité

On effectue un test booléen en utilisant l'opérateur infixe == qui désigne la relation d'égalité. La réponse du programme sera donc « vrai » ou « faux » (« 1 » ou « 0 » sur XCAS et « true » ou « false » sur CAML)

On peut créer une fonction qui testera si un nombre divise un autre.

Entrées : 2 entiers n et p

début

| (quotient entier n/p)× p == n

fin

Algorithme 16 : Test de divisibilité

On effectue un test booléen en utilisant l'opérateur infixe == qui désigne la relation d'égalité. La réponse du programme sera donc « vrai » ou « faux » (« 1 » ou « 0 » sur XCAS et « true » ou « false » sur CAML)

Avec XCAS

```
divise(p,n):={iquo(n,p)*p==n};;
```

Avec CAML

```
# let divise p n = ((n/p)*p=n);;
```

Qui s'utilise ainsi :

```
# divise 4 13;;
- : bool = false
# divise 4 12;;
- : bool = true
```

Avec CAML

```
# let divise p n = ((n/p)*p=n);;
```

Qui s'utilise ainsi :

```
# divise 4 13;;
- : bool = false
# divise 4 12;;
- : bool = true
```

Sommaire

Nous allons voir plusieurs algorithmes pour calculer le PGCD de deux entiers.

Le plus naturel est la version récursive de l'algorithme d'Euclide :

```
PGCD1(a,b) Entrées : 2 entiers a et b
début
    si b=0 alors
        retourner a
    sinon
        retourner PGCD1(b,reste(a,b))
fin
```

Algorithme 17 : Version récursive de l'algorithme d'Euclide

Nous allons voir plusieurs algorithmes pour calculer le PGCD de deux entiers.
Le plus naturel est la version récursive de l'algorithme d'Euclide :

```
PGCD1(a,b) Entrées : 2 entiers a et b
début
    si b=0 alors
        retourner a
    sinon
        retourner PGCD1(b,reste(a,b))
fin
```

Algorithme 18 : Version récursive de l'algorithme d'Euclide

Avec XCAS

```
PGCD1(a,b):={
if(b==0){return(a)}
else{return(PGCD1(b,irem(a,b)))}
}
```

Avec CAML

```
# let rec pgcd (a,b)
        =
        if b=0
         then a
         else  pgcd(b,a-a/b*b);;
```

Version impérative

En version impérative, c'est un peu plus long et moins naturel ;-)

Entrées : 2 entiers a et b
début
  tant que $b \neq 0$ **faire**
    r$\longleftarrow$ reste(a,b);
    a$\longleftarrow$ b;
    b$\longleftarrow$ r;
fin
retourner *a*

Algorithme 19 : PGCD : version impérative de l'algorithme d'Euclide

Avec XCAS

```
PGCD2(a,b):={
local r;
while(b!=0){
r:=irem(a,b);
a:=b;
b:=r;
}
return(a)
}:;
```

Algorithme des différences

On peut également utiliser le fait que $a \wedge b = a \wedge (b - a) = a \wedge (a - b)$, mais c'est plus (beaucoup trop) long... Une idée de recherche peut être de comparer les deux algorithmes.

```
PGCD3(a,b) Entrées : 2 entiers a et b
début
    si b=0 alors
        retourner a
    si a>=b alors
        retourner PGCD3(b,a-b)
    sinon
        retourner PGCD3(a,b-a)
fin
```

Algorithme 20 : PGCD par l'algorithme des différences en version récursive

Algorithme des différences

On peut également utiliser le fait que $a \wedge b = a \wedge (b - a) = a \wedge (a - b)$, mais c'est plus (beaucoup trop) long... Une idée de recherche peut être de comparer les deux algorithmes.

```
PGCD3(a,b) Entrées : 2 entiers a et b
début
    si b=0 alors
        retourner a
    si a>=b alors
        retourner PGCD3(b,a-b)
    sinon
        retourner PGCD3(a,b-a)
fin
```

Algorithme 21 : PGCD par l'algorithme des différences en version récursive

Avec XCAS

```
PGCD3(a,b):={
if(b==0){return(a)}
if(a>=b){return(PGCD3(b,a-b))}
else{return(PGCD3(a,b-a))}
}
```

Avec CAML

```
let rec pgcdif (a,b)
        =
        if b=0
         then a
         else  if a>b
                 then pgcdif(b,a-b)
         else pgcdif(a,b-a);;
```

Sommaire

Petit rappel sur cet algorithme qui permet de prouver le théorème de Bézout et d'obtenir des coefficients vérifiant $au + bv = a \wedge b$:

k	u_k	v_k	r_k	q_k
0	1	0	$r_0 = a$	/
1	0	1	$r_1 = b$	q_1
2	$u_0 - u_1 q_1$	$v_0 - v_1 q_1$	$r_2 = r_0 - r_1 q_1$	q_2
3	$u_1 - u_2 q_2$	$v_1 - v_2 q_2$	$r_3 = r_1 - r_2 q_2$	q_3
$\vdots$			$\vdots$	$\vdots$
$p-1$			$r_{p-1} = a \wedge b$	q_{p-1}
p			$r_p = 0$	

Petit rappel sur cet algorithme qui permet de prouver le théorème de Bézout et d'obtenir des coefficients vérifiant $au + bv = a \wedge b$:

k	u_k	v_k	r_k	q_k
0	1	0	$r_0 = a$	/
1	0	1	$r_1 = b$	q_1
2	$u_0 - u_1 q_1$	$v_0 - v_1 q_1$	$r_2 = r_0 - r_1 q_1$	q_2
3	$u_1 - u_2 q_2$	$v_1 - v_2 q_2$	$r_3 = r_1 - r_2 q_2$	q_3
$\vdots$			$\vdots$	$\vdots$
$p-1$			$r_{p-1} = a \wedge b$	q_{p-1}
p			$r_p = 0$	

Par exemple, pour 19 et 15 :

k	u_k	v_k	r_k	q_k	
0	1	0	19	/	L_0
1	0	1	15	1	L_1
2	1	-1	4	3	$L_2 \leftarrow L_0 - 1 \times L_1$
3	-3	4	3	1	$L_3 \leftarrow L_1 - 3 \times L_2$
4	$u = 4$	$v = -5$	$19 \wedge 15 = 1$	3	$L_4 \leftarrow L_2 - 1 \times L_3$
5			0		$L_5 \leftarrow L_3 - 3 \times L_4$

C'est-à-dire $4 \times 19 - 5 \times 15 = 1$ et $19 \wedge 15 = 1$.

Par exemple, pour 19 et 15 :

k	u_k	v_k	r_k	q_k	
0	1	0	19	/	L_0
1	0	1	15	1	L_1
2	1	-1	4	3	$L_2 \leftarrow L_0 - 1 \times L_1$
3	-3	4	3	1	$L_3 \leftarrow L_1 - 3 \times L_2$
4	$u = 4$	$v = -5$	$19 \wedge 15 = 1$	3	$L_4 \leftarrow L_2 - 1 \times L_3$
5			0		$L_5 \leftarrow L_3 - 3 \times L_4$

C'est-à-dire $4 \times 19 - 5 \times 15 = 1$ et $19 \wedge 15 = 1$.

Par exemple, pour 19 et 15 :

k	u_k	v_k	r_k	q_k	
0	1	0	19	/	L_0
1	0	1	15	1	L_1
2	1	-1	4	3	$L_2 \leftarrow L_0 - 1 \times L_1$
3	-3	4	3	1	$L_3 \leftarrow L_1 - 3 \times L_2$
4	$u = 4$	$v = -5$	$19 \wedge 15 = 1$	3	$L_4 \leftarrow L_2 - 1 \times L_3$
5			0		$L_5 \leftarrow L_3 - 3 \times L_4$

C'est-à-dire $4 \times 19 - 5 \times 15 = 1$ et $19 \wedge 15 = 1$.

Version récursive

En version récursive, on peut procéder en deux étapes :

aee(u,v,r,u',v',r');

Entrées : 6 entiers u, v, r, u', v', r'

début

  si *r'=0* **alors**

    retourner ([u,v,r])

  sinon

    aee(u',v',r',u-quotient(r,r')×u',v-quotient(r,r')×v',r-quotient(r,r')×r')

fin

AEE(a,b) **Entrées** : 2 entiers a et b

début

  aee(1,0,a,0,1,b)

fin

Avec XCAS

```
aee(u,v,r,u',v',r'):={
if(r'==0){return([u,v,r])};
aee(u',v',r',u-iquo(r,r')*u',v-iquo(r,r')*v',r-
    iquo(r,r')*r')
}:;

AEE(a,b):=aee(1,0,a,0,1,b):;
```

Par exemple, AEE(19,15) renvoie [4,−5,1] ce qui signifie que $4 \times 19 - 5 \times 15 = 1$ et que $19 \wedge 15 = 1$.

Avec XCAS

```
aee(u,v,r,u',v',r'):={
if(r'==0){return([u,v,r])};
aee(u',v',r',u-iquo(r,r')*u',v-iquo(r,r')*v',r-
    iquo(r,r')*r')
}:;

AEE(a,b):=aee(1,0,a,0,1,b):;
```

Par exemple, AEE(19,15) renvoie [4,-5,1] ce qui signifie que $4 \times 19 - 5 \times 15 = 1$ et que $19 \wedge 15 = 1$.

Avec CAML

```
# let rec aee(u,v,r,u',v',r')
    =
  if r'=0
   then [u;v;r]
  else aee(u',v',r',u-r/r'*u',v-r/r'*v',r-r/r'*r')
      ;;

# let aet(a,b)
      = aee(1,0,a,0,1,b);;
```

Version impérative

Entrées : 2 entiers a et b

Initialisation : liste a$\longleftarrow$ (1,0,a);

liste b$\longleftarrow$ (0,1,b);

reste$\longleftarrow$ b;

début

  tant que *reste*$\neq$*0* **faire**

    q$\longleftarrow$ quotient(3^{e}opérande de liste a, reste);

    liste temp$\longleftarrow$ liste a-q$\times$liste b;

    liste a$\longleftarrow$ liste b;

    liste b$\longleftarrow$ liste temp;

    reste$\longleftarrow$ 3^{e}opérande de liste b

fin

retourner *liste a*

Algorithme 23 : algorithme d'Euclide étendu : version impérative

Avec XCAS

```
bezout(a,b):={
local la,lb,lr,q,reste;
la:=[1,0,a];
lb:=[0,1,b];
reste:=b;
while (reste !=0){
q:=quotient(la[2],reste);
lr:=la+(-q)*lb;
la:=lb;
lb:=lr;
reste:=lb[2];
}
return(la);
}
```

Sommaire

- Vous connaissez l'algorithme suivant :

$$\begin{aligned} 172 &= 3 \times 51 + 19 && (1)\\ 51 &= 2 \times 19 + 13 && (2)\\ 19 &= 1 \times 13 + 6 && (3)\\ 13 &= 2 \times 6 + 1 && (4)\\ 6 &= 6 \times 1 + 0 && (5) \end{aligned}$$

- On peut donc facilement compléter la suite d'égalité suivante :

$$\frac{172}{51} = 3 + \frac{19}{51} = 3 + \frac{1}{\frac{51}{19}} = 3 + \frac{1}{2 + \frac{13}{19}} = \cdots$$

- Vous connaissez l'algorithme suivant :

$$\begin{aligned} 172 &= 3 \times 51 + 19 && (1)\\ 51 &= 2 \times 19 + 13 && (2)\\ 19 &= 1 \times 13 + 6 && (3)\\ 13 &= 2 \times 6 + 1 && (4)\\ 6 &= 6 \times 1 + 0 && (5) \end{aligned}$$

- On peut donc facilement compléter la suite d'égalité suivante :

$$\frac{172}{51} = 3 + \frac{19}{51} = 3 + \frac{1}{\frac{51}{19}} = 3 + \frac{1}{2 + \frac{13}{19}} = \dots$$

Algorithme récursif

On suppose connus les algorithmes donnant le reste et le quotient d'une division euclidienne.

```
fc(a,b)  début
  | si b=0 alors
  |   └ retourner liste vide
  | ;
fin
Retour[quotient(a,b),fc(b,reste(a,b))]
```

Algorithme 24 : fractions continues : version récursive

L'algorithme « réciproque » sera étudié plus loin **??**.

Avec XCAS

```
fc(a,b):={
if(b==0){return([])};
return(concat(iquo(a,b),fc(b,irem(a,b))))
};:;
```

Alors `fc(172,51)` renvoie `[3,2,1,2,6]`

Avec XCAS

```
fc(a,b):={
if(b==0){return([])};
return(concat(iquo(a,b),fc(b,irem(a,b))))
}:;
```

Alors `fc(172,51)` renvoie `[3,2,1,2,6]`

Avec CAML

```
# let rec fc (a,b)
        =
        if b = 0
         then []
         else  a/b :: fc(b,a-a/b*b);;
```

Algorithme impératif

Entrées : 2 entiers a et b
Initialisation : num$\longleftarrow$ a;
den$\longleftarrow$ b;
res$\longleftarrow$ reste(num,den);
Liste$\longleftarrow$ vide;
début
  tant que *res$\neq$0* **faire**
    Liste$\longleftarrow$ Liste,quotient(num,den);
    num$\longleftarrow$ den;
    den$\longleftarrow$ res;
    res$\longleftarrow$ reste(num,den);
fin
retourner *[Liste,quotient(num,den)]*

Algorithme 25 : Fractions continues : version impérative

Avec XCAS

```
frac_cont(a,b):={
local num,den,res,Liste;
num:=a;
den:=b;
res:=remain(num,den);
Liste:=NULL;
while(res>0){
    Liste:=Liste,iquo(num,den);
    num:=den;
    den:=res;
    res:=remain(num,den);
}
[Liste,iquo(num,den)];
}:;
```

Sommaire

Première version

Entrées : un entier n

début

  pour *k de 2 jusque partie entière de* $\sqrt{n}$ **faire**

    si *reste de la division de n par k est nul* **alors**

      retourner *n n'est pas premier*

  retourner *n est premier*

fin

Algorithme 26 : test de primalité 1

Avec XCAS

```
testPrem1(n):={
local k;
pour k de 2 jusque floor(sqrt(n)) faire
  si irem(n,k)==0 alors return(n+" n'est pas
     premier");
  fsi;
fpour;
return(n+" est premier");
}:;
```

```
testPrem1(n):={
local k;
 for(k:=2;k<=floor(sqrt(n));k++){
   if(irem(n,k)==0){return(n+" n'est pas premier")
      };
 };
```

Avec XCAS

```
testPrem1(n):={
local k;
pour k de 2 jusque floor(sqrt(n)) faire
  si irem(n,k)==0 alors return(n+" n'est pas
     premier");
  fsi;
fpour;
return(n+" est premier");
}:;
```

```
testPrem1(n):={
local k;
 for(k:=2;k<=floor(sqrt(n));k++){
   if(irem(n,k)==0){return(n+" n'est pas premier")
      };
 };
```

Avec XCAS

```
testPrem1(n):={
local k;
pour k de 2 jusque floor(sqrt(n)) faire
  si irem(n,k)==0 alors return(n+" n'est pas
     premier");
  fsi;
fpour;
return(n+" est premier");
}:;
```

```
testPrem1(n):={
local k;
 for(k:=2;k<=floor(sqrt(n));k++){
   if(irem(n,k)==0){return(n+" n'est pas premier")
      };
 };
```

Avec CAML

```
# let test_naif n =
      let rec test_local(m,k)=
          if m*m>k then "premier"
else if k/m*m =k then "compose"
          else test_local(m+1,k)
      in test_local(2,n);;
```

On note au passage comment utiliser une fonction localement à l'intérieur d'une autre. On aurait pu tout aussi bien les écrire séparément :

```
# let rec  test_local(m,k)=
          if m*m>k then "premier"
else if k/m*m =k then "compose"
          else test_local(m+1,k);;

# let test_naif n = test_local(2,n);;
```

Avec CAML

```
# let test_naif n =
      let rec test_local(m,k)=
          if m*m>k then "premier"
else if k/m*m =k then "compose"
          else test_local(m+1,k)
      in test_local(2,n);;
```

On note au passage comment utiliser une fonction localement à l'intérieur d'une autre. On aurait pu tout aussi bien les écrire séparément :

```
# let rec  test_local(m,k)=
          if m*m>k then "premier"
else if k/m*m =k then "compose"
          else test_local(m+1,k);;

# let test_naif n = test_local(2,n);;
```

Deuxième version

Une première possibilité pour diviser le nombre de tests par 2 est de vérifier au départ si le nombre est pair. Dans le cas contraire, on peut faire des sauts de 2 à partir de 3 :

Entrées : un entier n
début
  si *n est pair* **alors**
    retourner *n n'est pas premier*
  pour *k de 2 jusque partie entière de* $\sqrt{n}$ *avec un pas de 2* **faire**
    si *reste de la division de n par k est nul* **alors**
      retourner *n n'est pas premier*
  retourner *n est premier*
fin

Algorithme 27 : test de primalité 2

Deuxième version

Une première possibilité pour diviser le nombre de tests par 2 est de vérifier au départ si le nombre est pair. Dans le cas contraire, on peut faire des sauts de 2 à partir de 3 :

Entrées : un entier n
début
  si *n est pair* **alors**
    retourner *n n'est pas premier*
  pour *k de 2 jusque partie entière de* $\sqrt{n}$ *avec un pas de 2*
  faire
    si *reste de la division de n par k est nul* **alors**
      retourner *n n'est pas premier*
  retourner *n est premier*
fin

Algorithme 28 : test de primalité 2

Avec XCAS

```
testPrem1(n):={
local k;
si irem(n,2)==0 alors return(n+" n'est pas
    premier"); fsi;
pour k de 3 jusque floor(sqrt(n)) pas 2 faire
  si irem(n,k)==0 alors return(n+" n'est pas
     premier");
  fsi;
fpour;
return(n+" est premier");
};;
```

Avec XCAS en anglais

```
testPrem1(n):={
local k;
 if(irem(n,2)==0){return(n+" n'est pas premier")
     };
 for(k:=3; k<=floor(sqrt(n));k:=k+2){
     if(irem(n,k)==0){return(n+" n'est pas premier
         ")};
 };
return(n+" est premier");
};:;
```

Plus que 291 secondes... On a bien divisé le temps de compilation par deux.

Avec XCAS en anglais

```
testPrem1(n):={
local k;
 if(irem(n,2)==0){return(n+" n'est pas premier")
     };
 for(k:=3; k<=floor(sqrt(n));k:=k+2){
     if(irem(n,k)==0){return(n+" n'est pas premier
         ")};
 };
return(n+" est premier");
};:;
```

Plus que 291 secondes... On a bien divisé le temps de compilation par deux.

Avec CAML

```
#  let test_naif_2 n =
     let rec test_local(m,k)=
      if k/2*2=k then "pair"
      else
        if m*m>k then "premier"
else if k/m*m =k then "compose"
        else test_local(m+2,k)
     in test_local(3,n);;
```

On peut affiner en enlevant les multiples de 3.

Sommaire

Version impérative avec XCAS

```
erato(n):={
local j,k,P;
P:=[seq(k,k=1..n)];
P[0]:=0;
pour j de 2 jusque floor(sqrt(n)) faire
  si P[j-1]>=1 alors
    pour k de 2 jusque floor(n/j) faire
      P[j*k-1]:=0;
    fpour;
  fsi;
fpour;
return(select(x->(x>=1),P));
};;
```

En anglais

```
erato(n):={
local j,k,P;
P:=[seq(k,k=1..n)];
P[0]:=0;
for(j:=2;j<=floor(sqrt(n));j++){
  if(P[j-1]>=1){
    for(k:=2;k<=floor(n/j);k++){
      P[j*k-1]:=0;
    };
  };
};
return(select(x->(x>=1),P));
}:;
```

Avec CAML

En CAML, la récursion peut se faire en étapes pour plus de lisibilité.
On fabrique une fonction qui renvoie un intervalle entier dont on a entré les bornes :

```
# let rec  inter(a,b)= if a>b then [] else a::
   inter(a+1,b);;
```

Ainsi :

```
# inter(2,12);;
- : int list = [2; 3; 4; 5; 6; 7; 8; 9; 10; 11;
   12]
```

Avec CAML

En CAML, la récursion peut se faire en étapes pour plus de lisibilité.
On fabrique une fonction qui renvoie un intervalle entier dont on a entré les bornes :

```
# let rec  inter(a,b)= if a>b then [] else a::
   inter(a+1,b);;
```

Ainsi :

```
# inter(2,12);;
- : int list = [2; 3; 4; 5; 6; 7; 8; 9; 10; 11;
   12]
```

Avec CAML

En CAML, la récursion peut se faire en étapes pour plus de lisibilité.
On fabrique une fonction qui renvoie un intervalle entier dont on a entré les bornes :

```
# let rec  inter(a,b)= if a>b then [] else a::
   inter(a+1,b);;
```

Ainsi :

```
# inter(2,12);;
- : int list = [2; 3; 4; 5; 6; 7; 8; 9; 10; 11;
   12]
```

Avec CAML

Ensuite, on crée une fonction qui retire les multiples d'un entier dans une liste donnée :

```
# let rec retmultiples(l,n)=
        if l=[] then [] else
              if hd(l)/n*n<>hd(l) then hd(l)::(
                  retmultiples(tl(l),n)) else
                  retmultiples(tl(l),n);;
```

Ainsi on peut enlever les multiples de 3 compris entre 2 et 12 :

```
# retmultiples(inter(2,12),3);;
- : int list = [2; 4; 5; 7; 8; 10; 11]
```

Avec CAML

Ensuite, on crée une fonction qui retire les multiples d'un entier dans une liste donnée :

```
# let rec retmultiples(l,n)=
        if l=[] then [] else
              if hd(l)/n*n<>hd(l) then hd(l)::(
                  retmultiples(tl(l),n)) else
                  retmultiples(tl(l),n);;
```

Ainsi on peut enlever les multiples de 3 compris entre 2 et 12 :

```
# retmultiples(inter(2,12),3);;
- : int list = [2; 4; 5; 7; 8; 10; 11]
```

Avec CAML

Enfin, voici le cœur de la procédure :

```
# let crible(m)=
      let rec crible_rec(l,n)=
        if n>m then []
        else
          if n*n>m then n::(crible_rec(l,n+1))
  else n::retmultiples(crible_rec(l,n+1),n)
in crible_rec(inter(2,m),2);;
```

L'utilisation est simple :

```
# crible(100);;
- : int list =
[2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41;
   43; 47; 53; 59; 61; 67; 71;
 73; 79; 83; 89; 97]
```

Avec CAML

Enfin, voici le cœur de la procédure :

```
# let crible(m)=
      let rec crible_rec(l,n)=
        if n>m then []
        else
          if n*n>m then n::(crible_rec(l,n+1))
  else n::retmultiples(crible_rec(l,n+1),n)
in crible_rec(inter(2,m),2);;
```

L'utilisation est simple :

```
# crible(100);;
- : int list =
[2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41;
   43; 47; 53; 59; 61; 67; 71;
 73; 79; 83; 89; 97]
```

Avec CAML

Enfin, voici le cœur de la procédure :

```
# let crible(m)=
      let rec crible_rec(l,n)=
        if n>m then []
        else
          if n*n>m then n::(crible_rec(l,n+1))
  else n::retmultiples(crible_rec(l,n+1),n)
in crible_rec(inter(2,m),2);;
```

L'utilisation est simple :

```
# crible(100);;
- : int list =
[2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41;
   43; 47; 53; 59; 61; 67; 71;
 73; 79; 83; 89; 97]
```

Sommaire

On pense au logarithme ou à la rigueur à une ruse informatique du style :« je transforme mon nombre en chaîne de caractères et je demande au logiciel de compter le nombre de caractères de la chaîne (ce qu'ils savent tous faire) ».

La première solution est peu utilisable en 2^{nde} et la deuxième peu intéressante mathématiquement.

Alors on « pense base 10 » :

```
NbChiffres(n) Entrées : un entier naturel n
début
    si n<10 alors
        1
    sinon
        1+NbChiffres(quotient_entier(n,10))
fin
```

Algorithme 29 : nombre de chiffres d'un entier naturel

On pense au logarithme ou à la rigueur à une ruse informatique du style :« je transforme mon nombre en chaîne de caractères et je demande au logiciel de compter le nombre de caractères de la chaîne (ce qu'ils savent tous faire) ».
La première solution est peu utilisable en 2^{nde} et la deuxième peu intéressante mathématiquement.

Alors on « pense base 10 » :

```
NbChiffres(n) Entrées : un entier naturel n
début
    si n<10 alors
        1
    sinon
        1+NbChiffres(quotient_entier(n,10))
fin
```

Algorithme 30 : nombre de chiffres d'un entier naturel

On pense au logarithme ou à la rigueur à une ruse informatique du style :« je transforme mon nombre en chaîne de caractères et je demande au logiciel de compter le nombre de caractères de la chaîne (ce qu'ils savent tous faire) ».
La première solution est peu utilisable en 2^{nde} et la deuxième peu intéressante mathématiquement.
Alors on « pense base 10 » :

```
NbChiffres(n) Entrées : un entier naturel n
début
    si n<10 alors
        1
    sinon
        1+NbChiffres(quotient_entier(n,10))
fin
```

Algorithme 31 : nombre de chiffres d'un entier naturel

Avec XCAS

```
NbChiffres(n):={
if(n<10){1}else{1+NbChiffres(iquo(n,10))}
}:;
```

Avec CAML

```
# let rec NbChiffres n = if n<10 then 1 else 1+
    NbChiffres(n/10);;
```

Sommaire

Une méthode pour obtenir l'écriture en base 2 d'un nombre est d'effectuer des divisions successives. Par exemple pour 11 :

11	2
1	5

5	2
1	2

2	2
0	1

1	2
1	0

$$
\begin{aligned}
11 &= \left(2 \times 5 + 1\right) \\
&= \left(2 \times \left(2 \times 2 + 1\right) + 1\right) \\
&= \left(2 \times \left(2 \times \left(2 \times 1\right) + 1\right) + 1\right) \\
&= \left(2 \times \left(2^2 + 1\right) + 1\right) \\
&= 2^3 + 2 + 1 \\
&= 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0
\end{aligned}
$$

L'écriture de 11 en base 2 est donc 1011 : c'est la liste des restes obtenus mais dans l'ordre inverse.

Une méthode pour obtenir l'écriture en base 2 d'un nombre est d'effectuer des divisions successives. Par exemple pour 11 :

1	1	2		5	2	2	2	1	2	1	2
1		5		2		0		1		1	0

$$
\begin{aligned}
11 &= \left(2 \times 5 + 1\right) \\
&= \left(2 \times \left(2 \times 2 + 1\right) + 1\right) \\
&= \left(2 \times \left(2 \times \left(2 \times 1\right) + 1\right) + 1\right) \\
&= \left(2 \times \left(2^2 + 1\right) + 1\right) \\
&= 2^3 + 2 + 1 \\
&= 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0
\end{aligned}
$$

L'écriture de 11 en base 2 est donc 1011 : c'est la liste des restes obtenus mais dans l'ordre inverse.

Une méthode pour obtenir l'écriture en base 2 d'un nombre est d'effectuer des divisions successives. Par exemple pour 11 :

11	2
1	5

5	2
1	2

2	2
0	1

1	2
1	0

$$
\begin{aligned}
11 &= \left(2 \times 5 + 1\right) \\
&= \left(2 \times \left(2 \times 2 + 1\right) + 1\right) \\
&= \left(2 \times \left(2 \times \left(2 \times 1\right) + 1\right) + 1\right) \\
&= \left(2 \times \left(2^2 + 1\right) + 1\right) \\
&= 2^3 + 2 + 1 \\
&= 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0
\end{aligned}
$$

L'écriture de 11 en base 2 est donc 1011 : c'est la liste des restes obtenus mais dans l'ordre inverse.

Entrées : un entier n

Initialisation : $r \longleftarrow$ le reste de la division de n par 2;

$q \longleftarrow$ le quotient de la division de n par 2;

R une liste contenant r au départ.

début

  tant que $q>0$ **faire**

    r$\longleftarrow$ le reste de la division de q par 2;

    $q \longleftarrow$ le quotient de la division de q par 2;

    On ajoute r au début de la liste R.

  retourner R

fin

Algorithme 32 : décomposition en base 2

Avec XCAS

```
base_deux(n):={
local r,R,q;
r:=irem(n,2);
q:=iquo(n,2);
R:=[r];
tantque q>0 faire
 r:=irem(q,2);
 q:=iquo(q,2);
 R:=concat([r],R);
ftantque;
return(R);
};;
```

Version récursive

Notons q_n et r_n le quotient et le reste après n divisions.

q_{n+1} est le quotient entier de q_n par 2 et on rajoute r_{n+1} à notre liste de restes.

Après un certain nombre de divisions, le quotient sera strictement inférieur à 2.

```
Binaire(n) Entrées : un entier n
début
    si n<2 alors
        retourner n
    sinon
        retourner Binaire(quotient(n,2)),reste(n,2)
fin
```

Algorithme 33 : Conversion binaire : version récursive

Version récursive

Notons q_n et r_n le quotient et le reste après n divisions.
q_{n+1} est le quotient entier de q_n par 2 et on rajoute r_{n+1} à notre liste de restes.

Après un certain nombre de divisions, le quotient sera strictement inférieur à 2.

```
Binaire(n) Entrées : un entier n
début
    si n<2 alors
        retourner n
    sinon
        retourner Binaire(quotient(n,2)),reste(n,2)
fin
```

Algorithme 34 : Conversion binaire : version récursive

Version récursive

Notons q_n et r_n le quotient et le reste après n divisions.
q_{n+1} est le quotient entier de q_n par 2 et on rajoute r_{n+1} à notre liste de restes.
Après un certain nombre de divisions, le quotient sera strictement inférieur à 2.

```
Binaire(n) Entrées : un entier n
début
    si n<2 alors
        retourner n
    sinon
        retourner Binaire(quotient(n,2)),reste(n,2)
fin
```

Algorithme 35 : Conversion binaire : version récursive

Version récursive

Notons q_n et r_n le quotient et le reste après n divisions.
q_{n+1} est le quotient entier de q_n par 2 et on rajoute r_{n+1} à notre liste de restes.
Après un certain nombre de divisions, le quotient sera strictement inférieur à 2.

```
Binaire(n) Entrées : un entier n
début
    si n<2 alors
        retourner n
    sinon
        retourner Binaire(quotient(n,2)),reste(n,2)
fin
```

Algorithme 36 : Conversion binaire : version récursive

Avec XCAS

```
Binaire(n):={
    if (n<2){return n}
    else{
      return(Binaire(iquo(n,2)),irem(n,2));
    }
 };;
```

Avec CAML

```
# let rec binaire n =
        if n<2
          then [n]
        else  binaire(n/2) @ [n mod 2];;
```

Remarque

On utilise `liste 1 @ liste 2` pour concaténer deux listes et `élément :: liste` pour rajouter un élément en début de liste.

Avec CAML

```
# let rec binaire n =
        if n<2
          then [n]
        else  binaire(n/2) @ [n mod 2];;
```

Remarque

On utilise `liste 1 @ liste 2` pour concaténer deux listes et `élément :: liste` pour rajouter un élément en début de liste.

Sommaire

Sommaire

On travaillera avec CAML.
On aura besoin d'un algorithme de calcul du PGCD :

```
#  let rec pgcd (a,b) =
        if b=0
         then a
         else  pgcd(b,a-a/b*b);;
```

On va représenter une fraction par une liste du type [numérateur ;dénominateur].
Pour prévenir toute division par zéro on va créer un opérateur traitant les exceptions qu'on nommera `Division_par_zero` et qui nous servira de garde-fou :

```
# exception Division_par_zero;;
```

On invoquera cette exception avec la commande `raise` : étant donné le niveau d'anglais des élèves, cette commande est naturelle !...

On va représenter une fraction par une liste du type [numérateur ;dénominateur].
Pour prévenir toute division par zéro on va créer un opérateur traitant les exceptions qu'on nommera `Division_par_zero` et qui nous servira de garde-fou :

```
# exception Division_par_zero;;
```

On invoquera cette exception avec la commande `raise` : étant donné le niveau d'anglais des élèves, cette commande est naturelle !...

On peut créer un opérateur qui simplifie les fractions :

```
# let simp([a;b]) =
  if b=0  then raise Division_par_zero
  else [a/pgcd(a,b);b/pgcd(a,b)];;
```

Par exemple

```
# simp([18;12]);;
- : int list = [3; 2]
```

Mais

```
# simp([1;0]);;
Exception: Division_par_zero.
```

On peut créer un opérateur qui simplifie les fractions :

```
# let simp([a;b]) =
  if b=0  then raise Division_par_zero
  else [a/pgcd(a,b);b/pgcd(a,b)];;
```

Par exemple

```
# simp([18;12]);;
- : int list = [3; 2]
```

Mais

```
# simp([1;0]);;
Exception: Division_par_zero.
```

On peut créer un opérateur qui simplifie les fractions :

```
# let simp([a;b]) =
  if b=0  then raise Division_par_zero
  else [a/pgcd(a,b);b/pgcd(a,b)];;
```

Par exemple

```
# simp([18;12]);;
- : int list = [3; 2]
```

Mais

```
# simp([1;0]);;
Exception: Division_par_zero.
```

On crée ensuite une somme simplifiée de fractions :

```
# let som([a;b],[c;d]) =
  if b=0 or d=0 then  raise Division_par_zero else
  simp([a*d+b*c;b*d]);;
```

Par exemple :

```
# som([2;3],[1;6]);;
- : int list = [5; 6]
```

On crée ensuite une somme simplifiée de fractions :

```
# let som([a;b],[c;d]) =
  if b=0 or d=0 then  raise Division_par_zero else
  simp([a*d+b*c;b*d]);;
```

Par exemple :

```
# som([2;3],[1;6]);;
- : int list = [5; 6]
```

Une multiplication :

```
# let mul([a;b],[c;d]) =
  if b=0 or d=0 then raise Division_par_zero else
  simp([a*c;b*d]);;
```

Par exemple :

```
# mul([3;2],[2;9]);;
- : int list = [1; 3]
```

Une multiplication :

```
# let mul([a;b],[c;d]) =
  if b=0 or d=0 then raise Division_par_zero else
  simp([a*c;b*d]);;
```

Par exemple :

```
# mul([3;2],[2;9]);;
- : int list = [1; 3]
```

Un inverse :

```
#  let inv([a;b]) =
   if b=0 or a=0 then  raise Division_par_zero else
   simp([b;a]);;
```

Une division :

```
# let div([a;b],[c;d])=mul([a;b],inv([c;d]));;
```

Un inverse :

```
#  let inv([a;b]) =
   if b=0 or a=0 then  raise Division_par_zero else
   simp([b;a]);;
```

Une division :

```
# let div([a;b],[c;d])=mul([a;b],inv([c;d]));;
```

Par exemple, comment entrer

$$1 + \frac{2 + \frac{3}{4}}{1 - \frac{5}{6}}$$

```
# som([1;1],div(som([2;1],[3;4]),som([1;1],[-5;6])
    ));;
- : int list = [35; 2]
```

Évidemment, c'est moins pratique que de taper le résultat sur une machine mais ça permet de réfléchir aux différentes opérations, à la notion de fonction,...

Par exemple, comment entrer

$$1 + \frac{2 + \frac{3}{4}}{1 - \frac{5}{6}}$$

```
# som([1;1],div(som([2;1],[3;4]),som([1;1],[-5;6])
    ));;
- : int list = [35; 2]
```

Évidemment, c'est moins pratique que de taper le résultat sur une machine mais ça permet de réfléchir aux différentes opérations, à la notion de fonction,...

Par exemple, comment entrer

$$1 + \frac{2 + \frac{3}{4}}{1 - \frac{5}{6}}$$

```
# som([1;1],div(som([2;1],[3;4]),som([1;1],[-5;6])
    ));;
- : int list = [35; 2]
```

Évidemment, c'est moins pratique que de taper le résultat sur une machine mais ça permet de réfléchir aux différentes opérations, à la notion de fonction,...

Sommaire

Nous avions vu à l'algorithme ?? comment, à partir d'un nombre écrit sous forme fractionnaire, on pouvait obtenir son écriture sous forme de fraction continue.

Nous allons à présent voir comment effectuer l'opération inverse, c'est-à-dire comment obtenir une fraction irréductible à partir de la liste du développement en fraction continue.

Souvenez-vous :

$$\frac{172}{51} = 3 + \frac{19}{51} = 3 + \frac{1}{\frac{51}{19}} = 3 + \frac{1}{2 + \frac{13}{19}} = 3 + \frac{1}{2 + \frac{1}{1 + \frac{1}{2 + \frac{1}{6}}}} = [3; 2; 1; 2; 6]$$

Nous avions vu à l'algorithme **??** comment, à partir d'un nombre écrit sous forme fractionnaire, on pouvait obtenir son écriture sous forme de fraction continue.
Nous allons à présent voir comment effectuer l'opération inverse, c'est-à-dire comment obtenir une fraction irréductible à partir de la liste du développement en fraction continue.

Souvenez-vous :

$$\frac{172}{51} = 3 + \frac{19}{51} = 3 + \frac{1}{\frac{51}{19}} = 3 + \frac{1}{2 + \frac{13}{19}} = 3 + \frac{1}{2 + \frac{1}{1 + \frac{1}{2 + \frac{1}{6}}}} = [3; 2; 1; 2; 6]$$

Nous avions vu à l'algorithme **??** comment, à partir d'un nombre écrit sous forme fractionnaire, on pouvait obtenir son écriture sous forme de fraction continue.
Nous allons à présent voir comment effectuer l'opération inverse, c'est-à-dire comment obtenir une fraction irréductible à partir de la liste du développement en fraction continue.

Souvenez-vous :

$$\frac{172}{51} = 3 + \frac{19}{51} = 3 + \frac{1}{\frac{51}{19}} = 3 + \frac{1}{2 + \frac{13}{19}} = 3 + \frac{1}{2 + \frac{1}{1 + \frac{1}{2 + \frac{1}{6}}}} = [3; 2; 1; 2; 6]$$

L'algorithme

`frac_inv(lx);`

Entrées : La liste lx du développement en fraction continue

début

  si *lx ne contient qu'un seul élément* **alors**

    retourner *cet élément*

  sinon

    retourner *le premier élément de lx +l'inverse de* `frac_inv`*(lx privé de son premier élément)*

fin

Algorithme 37 : Fractions continues : le retour récursif

```
# let rec frac_inv(lx)=
      if length lx=1 then [hd(lx);1]
      else som([hd(lx);1],inv(frac_inv(tl(lx))));;
```

L'algorithme

`frac_inv(lx);`

Entrées : La liste lx du développement en fraction continue

début

  si *lx ne contient qu'un seul élément* **alors**

    retourner *cet élément*

  sinon

    retourner *le premier élément de lx +l'inverse de* `frac_inv`*(lx privé de son premier élément)*

fin

Algorithme 38 : Fractions continues : le retour récursif

```
# let rec frac_inv(lx)=
      if length lx=1 then [hd(lx);1]
      else som([hd(lx);1],inv(frac_inv(tl(lx))));;
```

Et voici comment l'utiliser :

```
# frac_inv([3;2;1;2;6]);;
- : int list = [172; 51]
```

Une petite curiosité au passage :

```
# frac_inv([0;3;2;1;2;6]);;
- : int list = [51; 172]
```

Et voici comment l'utiliser :

```
# frac_inv([3;2;1;2;6]);;
- : int list = [172; 51]
```

Une petite curiosité au passage :

```
# frac_inv([0;3;2;1;2;6]);;
- : int list = [51; 172]
```

Sommaire

Sommaire

Prenons l'exemple de $P(x) = 3x^5 - 2x^4 + 7x^3 + 2x^2 + 5x - 3$. Le calcul classique nécessite 5 additions et 15 multiplications.

On peut faire pas mal d'économies de calcul en suivant le schéma suivant :

$$\begin{aligned} P(x) &= \underbrace{a_n x^n + \cdots + a_2 x^2 + a_1 x}_{\text{on met } x \text{ en facteur}} + a_0 \\ &= \left(\underbrace{a_n x^{n-1} + \cdots + a_2 x}_{\text{on met } x \text{ en facteur}} + a_1 \right) x + a_0 \\ &= \cdots \\ &= (\cdots(((a_n x + a_{n-1})x + a_{n-2})x + a_{n-3})x + \cdots)\, x + a_0 \end{aligned}$$

Ici cela donne $P(x) = (((((3x) - 2)x + 7)x + 2)x + 5)x - 3$ c'est-à-dire 5 multiplications et 5 additions. En fait il y a au maximum 2×degré de P opérations (voire moins avec les zéros).

Prenons l'exemple de $P(x) = 3x^5 - 2x^4 + 7x^3 + 2x^2 + 5x - 3$. Le calcul classique nécessite 5 additions et 15 multiplications.
On peut faire pas mal d'économies de calcul en suivant le schéma suivant :

$$\begin{aligned} P(x) &= \underbrace{a_n x^n + \cdots + a_2 x^2 + a_1 x}_{\text{on met } x \text{ en facteur}} + a_0 \\ &= \left(\underbrace{a_n x^{n-1} + \cdots + a_2 x}_{\text{on met } x \text{ en facteur}} + a_1 \right) x + a_0 \\ &= \cdots \\ &= (\cdots(((a_n x + a_{n-1})x + a_{n-2})x + a_{n-3})x + \cdots)\, x + a_0 \end{aligned}$$

Ici cela donne $P(x) = (((((3x) - 2)x + 7)x + 2)x + 5)x - 3$ c'est-à-dire 5 multiplications et 5 additions. En fait il y a au maximum 2×degré de P opérations (voire moins avec les zéros).

Prenons l'exemple de $P(x) = 3x^5 - 2x^4 + 7x^3 + 2x^2 + 5x - 3$. Le calcul classique nécessite 5 additions et 15 multiplications.
On peut faire pas mal d'économies de calcul en suivant le schéma suivant :

$$\begin{aligned} P(x) &= \underbrace{a_n x^n + \cdots + a_2 x^2 + a_1 x}_{\text{on met } x \text{ en facteur}} + a_0 \\ &= \left(\underbrace{a_n x^{n-1} + \cdots + a_2 x}_{\text{on met } x \text{ en facteur}} + a_1 \right) x + a_0 \\ &= \cdots \\ &= (\cdots(((a_n x + a_{n-1})x + a_{n-2})x + a_{n-3})x + \cdots)\, x + a_0 \end{aligned}$$

Ici cela donne $P(x) = (((((3x) - 2)x + 7)x + 2)x + 5)x - 3$ c'est-à-dire 5 multiplications et 5 additions. En fait il y a au maximum 2×degré de P opérations (voire moins avec les zéros).

Algorithme

Il faut connaître les coefficients de P.

horner(P,x) **Entrées** : les coefficients a_i dans l'ordre décroissant des exposants des monômes, le degré n de P et le nombre x

début

  si *P est vide* **alors**

    retourner *0*

  sinon

    retourner *Tête de P* $+ x\times$ *horner(P privé de sa tête, x)*

fin

Algorithme 39 : algorithme de Hörner : version récursive

Avec XCAS

```
Horner(P,x):={
if(P=[]){return(0)}
else{return(P[0]+x*Horner(tail(P),x))}
}:;
```

Avec CAML

```
# let rec horner(p,x) =
    if p=[] then 0
    else   List.hd(p)+x*(horner(List.tl(p),x));;
```

Sommaire

Étudions une méthode d'exponentiation rapide.

Voyons comment procéder sur a^{11} :

$$a^{11} = a \cdot a^{10} = a \cdot \left(a^5\right)^2 = a \cdot \left(a \cdot a^4\right)^2 = a \cdot \left(a \cdot \left(a^2\right)^2\right)^2$$

Étudions une méthode d'exponentiation rapide.
Voyons comment procéder sur a^{11} :

$$a^{11} = a \cdot a^{10} = a \cdot \left(a^5\right)^2 = a \cdot \left(a \cdot a^4\right)^2 = a \cdot \left(a \cdot \left(a^2\right)^2\right)^2$$

Entrées : un réel a et un entier n

Initialisation : res$\longleftarrow$ 1;

puissance$\longleftarrow$ n;

temp$\longleftarrow$ a

début

  tant que *puissance non nulle* **faire**

    si *puissance est impaire* **alors**

      res$\longleftarrow$ temp$\times$res;

      puissance$\longleftarrow$ puissance $-$ 1

    puissance$\longleftarrow$ puissance $\div$ 2;

    temp$\longleftarrow$ temp$\times$temp

fin

Algorithme 40 : exponentiation rapide sans utiliser la base 2

Avec XCAS

```
expo_rapido(a,n):={
local res,temp,Puissance;
res:=1;
Puissance:=n;
temp:=a;
tantque Puissance >0 faire
    si irem(Puissance,2)==1 alors
        res:=temp*res;
        Puissance:=Puissance-1;
    fsi;
Puissance:=Puissance/2;
temp:=temp*temp;
ftantque;
return(res);
};;
```

Avec CAML

Avec CAML en version récursive c'est immédiat...

```
# let rec pow_rap(a,n)=
          if n=0 then a
  else if n/2*2=n then pow_rap(a,n/2)*pow_rap(a,n
     /2)
                 else a*pow_rap(a,n/2);;
```

Sommaire

Sommaire

On repère graphiquement qu'une fonction semble atteindre par exemple un minimum. Pour obtenir une approximation de ce minimum, on fournit un point de départ et une précision.

Version récursive avec XCAS

```
min_rec(f,xo,h):={
if(f(xo+h)>f(xo)){return(xo)}
else{min_rec(f,xo+h,h)}
}
```

Version récursive avec CAML

```
# let rec min_rec(f,xo,h)=
    if f(xo+.h)>f(xo) then xo
    else min_rec(f,xo+.h,h);;
```

Version impérative

Entrées : fonction f, point de départ xo et incrément i

Initialisation : Y⟵ f(xo);

X⟵ xo+i;

;

début

  tant que *t(X)<Y* **faire**

    Y⟵ evalf(t(X));

    X⟵ X+i;

  retourner *X-i*

fin

Algorithme 41 : approximation d'un minimum

Version impérative avec XCAS

```
minimum(f,xo,p):={
local X,Y,compteur;
Y:=f(xo);
X:=xo+10^(-p);
compteur:=0 // on rajoute un compteur pour que la
    compilation ne dure pas trop longtemps
tantque evalf(f(X))<Y et compteur<10^p faire
    Y:=evalf(f(X));
    X:=X+10^(-p);
    compteur:=compteur+1;
ftantque:;
 X-10^(-p);
}:;
```

Par exemple, pour la fonction $x \mapsto \sqrt{x^2+25}+\sqrt{(x-18)^2+49}$, il semble que le minimum soit vers 7.

Demandons une approximation du minimum à 10^{-5} près :

```
minimum(x->sqrt((x)^2+25)+sqrt((x-18)^2+49),7.0,5)
```

et on obtient 7.50000

Par exemple, pour la fonction $x \mapsto \sqrt{x^2+25} + \sqrt{(x-18)^2+49}$, il semble que le minimum soit vers 7.
Demandons une approximation du minimum à 10^{-5} près :

```
minimum(x->sqrt((x)^2+25)+sqrt((x-18)^2+49),7.0,5)
```

et on obtient 7.50000

Par exemple, pour la fonction $x \mapsto \sqrt{x^2+25} + \sqrt{(x-18)^2+49}$, il semble que le minimum soit vers 7.
Demandons une approximation du minimum à 10^{-5} près :

```
minimum(x->sqrt((x)^2+25)+sqrt((x-18)^2+49),7.0,5)
```

et on obtient 7.50000

Sommaire

Pour résoudre une équation du type $f(x) = 0$, on recherche graphiquement un intervalle $[a, b]$ où la fonction semble changer de signe.

On note ensuite m le milieu du segment $[a, b]$. On évalue le signe de $f(m)$. Si c'est le même que celui de $f(a)$ on remplace a par m et on recommence. Sinon, c'est b qu'on remplace et on recommence jusqu'à obtenir la précision voulue.

Pour résoudre une équation du type $f(x) = 0$, on recherche graphiquement un intervalle $[a, b]$ où la fonction semble changer de signe.
On note ensuite m le milieu du segment $[a, b]$. On évalue le signe de $f(m)$.
Si c'est le même que celui de $f(a)$ on remplace a par m et on recommence.
Sinon, c'est b qu'on remplace et on recommence jusqu'à obtenir la précision voulue.

Pour résoudre une équation du type $f(x) = 0$, on recherche graphiquement un intervalle $[a, b]$ où la fonction semble changer de signe.
On note ensuite m le milieu du segment $[a, b]$. On évalue le signe de $f(m)$.
Si c'est le même que celui de $f(a)$ on remplace a par m et on recommence.
Sinon, c'est b qu'on remplace et on recommence jusqu'à obtenir la précision voulue.

Version récursive avec XCAS

```
dicho_rec(f,a,b,eps):={
if(evalf(b-a)<eps){return(0.5*(b+a))};
if(f(a)*f(0.5*(b+a))>0){return(dicho(f,0.5*(b+a),b
    ,eps))}
                    else{return(dicho(f,a,0.5*(b+a)
                        ,eps))}
};:;
```

Version récursive avec CAML

```
# let rec dicho_rec(f,a,b,eps)=
    if abs_float(b-.a)<eps then 0.5*.(b+.a)
    else if f(a)*.f(0.5*.(b+.a))>0. then dicho_rec
       (f,0.5*.(b+.a),b,eps)
         else  dicho_rec(f,a,0.5*.(b+.a),eps);;
```

ce qui donne :

```
# dicho_rec( (fun x->(x*.x-.2.)),1.,2.,0.00001);;
- : float = 1.4142112731933594
```

Version récursive avec CAML

```
# let rec dicho_rec(f,a,b,eps)=
    if abs_float(b-.a)<eps then 0.5*.(b+.a)
    else if f(a)*.f(0.5*.(b+.a))>0. then dicho_rec
       (f,0.5*.(b+.a),b,eps)
         else  dicho_rec(f,a,0.5*.(b+.a),eps);;
```

ce qui donne :

```
# dicho_rec( (fun x->(x*.x-.2.)),1.,2.,0.00001);;
- : float = 1.4142112731933594
```

Version impérative

Entrées : une fonction f, les bornes a et b, une précision p

Initialisation : aa$\longleftarrow$ a, bb$\longleftarrow$ b

début

  tant que *bb-aa>p* **faire**

    si *signe de f((aa+bb)/2) est le même que celui de f(bb)*

    alors

      bb$\longleftarrow$ (aa+bb)/2

    sinon

      aa$\longleftarrow$ (aa+bb)/2

  retourner *(aa+bb)/2*

fin

Algorithme 42 : dichotomie

Version impérative avec XCAS

Avec XCAS, ne pas oublier de régler quelques problèmes de précision :

```
dicho(F,p,a,b):={
local aa,bb,k,f;
aa:=a;bb:=b;epsilon:=1e-100;
f:=unapply(F,x);k:=0;
 tantque evalf(bb-aa,p)>10^(-p) faire
    si sign(evalf(f((bb+aa)/2),p))==sign(evalf(f(
       bb),p))
      alors bb:=evalf((aa+bb)/2,p);
      sinon aa:=evalf((aa+bb)/2,p);
      k:=k+1; // un tour de plus au compteur
    fsi;
 ftantque;
return evalf((bb+aa)/2,p)+" est la solution
   trouvee apres "+k+" iterations";
};;
```

Sommaire

On veut calculer une intégrale d'une fonction par la méthode des rectangles. La définition récursive est assez naturelle.

```
# let rec integ(f,a,b,dx)=
      if a>b then 0.
      else f(a)*.dx+.integ(f,a+.dx,b,dx);;
```

Par exemple :

```
# let g(x)=x;;

# integ(g,0.,1.,0.0001);;
```

donne :

```
- : float = 0.50004999999960249
```

C'est-à-dire $\int_0^1 x\,dx \approx 0.5$

```
# let rec integ(f,a,b,dx)=
      if a>b then 0.
      else f(a)*.dx+.integ(f,a+.dx,b,dx);;
```

Par exemple :

```
# let g(x)=x;;

# integ(g,0.,1.,0.0001);;
```

donne :

```
- : float = 0.50004999999960249
```

C'est-à-dire $\int_0^1 x\mathrm{d}x \approx 0.5$

```
# let rec integ(f,a,b,dx)=
      if a>b then 0.
      else f(a)*.dx+.integ(f,a+.dx,b,dx);;
```

Par exemple :

```
# let g(x)=x;;

# integ(g,0.,1.,0.0001);;
```

donne :

```
- : float = 0.50004999999960249
```

C'est-à-dire $\int_0^1 x\mathrm{d}x \approx 0.5$

Sommaire

Soit un système $\begin{cases} a_{11}x + a_{12}y = b_1 \\ a_{21}x + a_{22}y = b_2 \end{cases}$.

Soit D le déterminant du sytème : $D = a_{11}a_{22} - a_{21}a_{12}$.

S'il est non nul, il est assez aisé de montrer en Seconde que les solutions sont :

$$x = \frac{b_1 a_{22} - b_2 a_{12}}{D} \qquad y = \frac{a_{11} b_2 - a_{21} b_1}{D}$$

Cela peut mettre en évidence les rôles différents joués par les inconnues (qui n'apparaissent pas en entrée dans l'algorithme) et les coefficients.

Soit un système $\begin{cases} a_{11}x + a_{12}y = b_1 \\ a_{21}x + a_{22}y = b_2 \end{cases}$.

Soit D le déterminant du sytème : $D = a_{11}a_{22} - a_{21}a_{12}$.

S'il est non nul, il est assez aisé de montrer en Seconde que les solutions sont :

$$x = \frac{b_1 a_{22} - b_2 a_{12}}{D} \qquad y = \frac{a_{11} b_2 - a_{21} b_1}{D}$$

Cela peut mettre en évidence les rôles différents joués par les inconnues (qui n'apparaissent pas en entrée dans l'algorithme) et les coefficients.

Soit un système $\begin{cases} a_{11}x + a_{12}y = b_1 \\ a_{21}x + a_{22}y = b_2 \end{cases}$.

Soit D le déterminant du sytème : $D = a_{11}a_{22} - a_{21}a_{12}$.

S'il est non nul, il est assez aisé de montrer en Seconde que les solutions sont :

$$x = \frac{b_1 a_{22} - b_2 a_{12}}{D} \qquad y = \frac{a_{11} b_2 - a_{21} b_1}{D}$$

Cela peut mettre en évidence les rôles différents joués par les inconnues (qui n'apparaissent pas en entrée dans l'algorithme) et les coefficients.

Soit un système $\begin{cases} a_{11}x + a_{12}y = b_1 \\ a_{21}x + a_{22}y = b_2 \end{cases}$.

Soit D le déterminant du sytème : $D = a_{11}a_{22} - a_{21}a_{12}$.

S'il est non nul, il est assez aisé de montrer en Seconde que les solutions sont :

$$x = \frac{b_1 a_{22} - b_2 a_{12}}{D} \qquad y = \frac{a_{11} b_2 - a_{21} b_1}{D}$$

Cela peut mettre en évidence les rôles différents joués par les inconnues (qui n'apparaissent pas en entrée dans l'algorithme) et les coefficients.

Entrées : Les coefficients

;

début

  si $D=0$ **alors**

    pas de solution unique

  sinon

    $x = \frac{b_1 a_{22} - b_2 a_{12}}{D}$ $\qquad$ $y = \frac{a_{11} b_2 - a_{21} b_1}{D}$

fin

Algorithme 43 : système de Cramer

Avec XCAS

```
cramer(a11,a12,b1,a21,a22,b2):={
 si a11*a22-a21*a12==0
   alors return("Pas de solution unique")
   sinon return((b1*a22-b2*a12)/(a11*a22-a21*a12),
        (a11*b2-a21*b1)/(a11*a22-a21*a12))
 fsi
};;
```

Sommaire

Héron d'Alexandrie n'avait pas attendu Newton et le calcul différentiel pour trouver une méthode permettant de déterminer une approximation de la racine carrée d'un nombre positif puisqu'il a vécu seize siècles avant Sir Isaac.

Si x_n est une approximation strictement positive par défaut de $\sqrt{a}$, alors a/x_n est une approximation par excès de $\sqrt{a}$ (pourquoi ?) et vice-versa.

La moyenne arithmétique de ces deux approximations est $\frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$ et constitue une meilleure approximation que les deux précédentes.

On peut montrer c'est une approximation par excès (en développant $\left(x_n - \sqrt{a}\right)^2$ par exemple).

On obtient naturellement un algorithme.

Héron d'Alexandrie n'avait pas attendu Newton et le calcul différentiel pour trouver une méthode permettant de déterminer une approximation de la racine carrée d'un nombre positif puisqu'il a vécu seize siècles avant Sir Isaac.

Si x_n est une approximation strictement positive par défaut de $\sqrt{a}$, alors a/x_n est une approximation par excès de $\sqrt{a}$ (pourquoi ?) et vice-versa.

La moyenne arithmétique de ces deux approximations est $\frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$ et constitue une meilleure approximation que les deux précédentes.

On peut montrer c'est une approximation par excès (en développant $\left(x_n - \sqrt{a}\right)^2$ par exemple).

On obtient naturellement un algorithme.

Héron d'Alexandrie n'avait pas attendu Newton et le calcul différentiel pour trouver une méthode permettant de déterminer une approximation de la racine carrée d'un nombre positif puisqu'il a vécu seize siècles avant Sir Isaac.

Si x_n est une approximation strictement positive par défaut de $\sqrt{a}$, alors a/x_n est une approximation par excès de $\sqrt{a}$ (pourquoi ?) et vice-versa.

La moyenne arithmétique de ces deux approximations est $\frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$ et constitue une meilleure approximation que les deux précédentes.

On peut montrer c'est une approximation par excès (en développant $\left(x_n - \sqrt{a}\right)^2$ par exemple).

On obtient naturellement un algorithme.

Héron d'Alexandrie n'avait pas attendu Newton et le calcul différentiel pour trouver une méthode permettant de déterminer une approximation de la racine carrée d'un nombre positif puisqu'il a vécu seize siècles avant Sir Isaac.

Si x_n est une approximation strictement positive par défaut de $\sqrt{a}$, alors a/x_n est une approximation par excès de $\sqrt{a}$ (pourquoi ?) et vice-versa.

La moyenne arithmétique de ces deux approximations est $\frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$ et constitue une meilleure approximation que les deux précédentes.

On peut montrer c'est une approximation par excès (en développant $\left(x_n - \sqrt{a}\right)^2$ par exemple).

On obtient naturellement un algorithme.

Héron d'Alexandrie n'avait pas attendu Newton et le calcul différentiel pour trouver une méthode permettant de déterminer une approximation de la racine carrée d'un nombre positif puisqu'il a vécu seize siècles avant Sir Isaac.

Si x_n est une approximation strictement positive par défaut de $\sqrt{a}$, alors a/x_n est une approximation par excès de $\sqrt{a}$ (pourquoi ?) et vice-versa.

La moyenne arithmétique de ces deux approximations est $\frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$ et constitue une meilleure approximation que les deux précédentes.

On peut montrer c'est une approximation par excès (en développant $\left(x_n - \sqrt{a}\right)^2$ par exemple).

On obtient naturellement un algorithme.

Entrées : le réel positif a, une première approximation par défaut strictement positive xo et une précision eps

Initialisation : X$\longleftarrow$ 0.5(xo+a/xo)

début

  tant que $|xo-a/xo|<eps$ **faire**

    X$\longleftarrow$ 0.5(X+a/X)

  retourner X

fin

Algorithme 44 : approximation de $\sqrt{a}$

Avec XCAS

```
heron(a,xo,eps):={
local X;
X:=0.5*(xo+a/xo);
while(abs(evalf(X-a/X))>=eps){
  X:=0.5*(X+a/X);
}
return(X)
}:;
```

Version récursive avec CAML

```
# let rec heron(a,xo,eps)=
      if abs_float(xo-.a/.xo)<eps then xo
      else heron(a,0.5*.(xo+.a/.xo),eps);;
```

Sommaire

Sommaire

On lance trois fois de suite une pièce de monnaie. On compte combien de fois pile (ou face) tombe.

Avec XCAS

- `rand(k)` renvoie un entier aléatoirement compris entre 0 et $k-1$;
- `count_eq(entier,liste)` compte le nombre de fois ou `entier` apparaît dans `liste`

Avec XCAS

- `rand(k)` renvoie un entier aléatoirement compris entre 0 et $k-1$;
- `count_eq(entier,liste)` compte le nombre de fois ou `entier` apparaît dans `liste`

Avec XCAS

```
piece(n):={
T:=NULL;
for(k:=1;k<=n;k++){
X:=0;
if(rand(2)==1){X:=X+1}
if(rand(2)==1){X:=X+1}
if(rand(2)==1){X:=X+1}
T:=T,X;
}
return(evalf([count_eq(0,[T])/n,count_eq(1,[T])/n,
    count_eq(2,[T])/n,count_eq(3,[T])/n])
)}:;
```

Avec XCAS

Par exemple :

```
piece(5000)
```

renvoie [0.1262,0.387,0.366,0.1208].

En effet, on peut vérifier avec `binomial(n,k,p)` qui renvoie $\binom{n}{k} p^k (1-p)^{n-k}$:

```
[seq(binomial(3,k,0.5),k=0..3)]
```

renvoie [0.125,0.375,0.375,0.125]

Avec XCAS

Par exemple :

```
piece(5000)
```

renvoie `[0.1262,0.387,0.366,0.1208]`.
En effet, on peut vérifier avec `binomial(n,k,p)` qui renvoie $\binom{n}{k}p^k(1-p)^{n-k}$:

```
[seq(binomial(3,k,0.5),k=0..3)]
```

renvoie `[0.125,0.375,0.375,0.125]`

Avec XCAS

On peut faire la moyenne sur une dizaine de séries de 5000 séries de 3 lancers :

```
mean([seq(piece(5000),k=0..10)])
```

et on obtient

[0.1272909091,0.3854,0.3637272727,0.1235818182]

Avec XCAS

On peut faire la moyenne sur une dizaine de séries de 5000 séries de 3 lancers :

```
mean([seq(piece(5000),k=0..10)])
```

et on obtient
[0.1272909091,0.3854,0.3637272727,0.1235818182]

Sommaire

On dispose de trois urnes, la première contenant 7 boules blanches et 4 noires, la deuxième 5 blanches et 2 noires, la troisième 6 blanches et 3 noires.
On tire une boule dans chaque urne et on note le nombre de boules blanches obtenues.
On adapte l'algorithme précédent.

```
boules(n):={
T:=NULL;
for(k:=1;k<=n;k++){
X:=0;
if(rand(11)<7){X:=X+1}
if(rand(7)<5){X:=X+1}
if(rand(9)<6){X:=X+1}
T:=T,X;
}
return(evalf([count_eq(0,[T])/n,count_eq(1,[T])/n,
   count_eq(2,[T])/n,count_eq(3,[T])/n])
)}:;
```

On obtient
$[0.0358, 0.2253818182, 0.4453272727, 0.2934909091]$
Or, par exemple, $\frac{7}{11} \times \frac{5}{7} \times \frac{6}{9} \approx 0,303$ et $\frac{4}{11} \times \frac{2}{7} \times \frac{3}{9} \approx 0,034$.

Sommaire

Cosme II de Médicis (Florence 1590-1621), Duc de Toscane, fut le protecteur de l'illustre Gallilée (né à Pise le 15 février 1564 et mort à Florence le 8 janvier 1642) son ancien précepteur. Profitant d'un moment de répit du savant entre l'écriture d'un théorème sur la chute des corps et la création de la lunette astronomique, le Grand Duc lui soumet le problème suivant : il a observé qu'en lançant trois dés cubiques et en faisant la somme des numéros des faces, on obtient plus souvent 10 que 9, alors qu'il y a autant de façons d'obtenir 9 que 10, à savoir six.
Après quelques réflexions, Galilée rédigea un petit mémoire sur les jeux de hasard en 1620 expliquant le phénomène.

On utilise `ranm(lignes,colonnes,'expérience')` qui renvoie une matrice `lignes`×`colonnes` où chaque terme est un résultat de `expérience`.

```
toscane(n):={
T:=ranm(1,n,'rand(6)+rand(6)+rand(6)+3');
return("Obtenir 9 :"+count_eq(9,T)*100.0/n+"%","
   Obtenir 10 :"+count_eq(10,T)*100.0/n+"%")
};;
```

Alors, pour 100 000 tirages, on obtient `"Obtenir 9 :11.263%","Obtenir 10 :12.535%"`

On utilise `ranm(lignes,colonnes,'expérience')` qui renvoie une matrice `lignes`×`colonnes` où chaque terme est un résultat de `expérience`.

```
toscane(n):={
T:=ranm(1,n,'rand(6)+rand(6)+rand(6)+3');
return("Obtenir 9 :"+count_eq(9,T)*100.0/n+"%","
    Obtenir 10 :"+count_eq(10,T)*100.0/n+"%")
};;
```

Alors, pour 100 000 tirages, on obtient `"Obtenir 9 :11.263%","Obtenir 10 :12.535%"`

On utilise `ranm(lignes,colonnes,'expérience')` qui renvoie une matrice `lignes`×`colonnes` où chaque terme est un résultat de `expérience`.

```
toscane(n):={
T:=ranm(1,n,'rand(6)+rand(6)+rand(6)+3');
return("Obtenir 9 :"+count_eq(9,T)*100.0/n+"%","
    Obtenir 10 :"+count_eq(10,T)*100.0/n+"%")
}:;
```

Alors, pour 100 000 tirages, on obtient `"Obtenir 9 :11.263%","Obtenir 10 :12.535%"`

Sommaire

On lance 10 fois de suite une pièce de monnaie et on s'intéresse au nombre maximal de résultats consécutifs égaux. On crée un programme qui simule autant de séries de lancers que l'on désire. C'est un peu plus compliqué à écrire...

```
piece_max(essais):={
local S,k,P,j,H,m,M,s,p,h;
S:=NULL;
 for(k:=1;k<=essais;k++){
  s:=NULL;
  P:=[seq(rand(2),m=0..9)];
  p:=0;
   while(p<9){
    j:=p+1;
     while (P[j]==P[p] and j<9){j:=j+1}
    s:=s,j-p;
    p:=j;
   }
  s:=SortD([s]);
  h:=head(s);
  S:=S,h;
 }
m:=evalf(mean([S]));
return("Sur "+essais+" series de 10 lancers, la moyenne du
    nombre maximal de resultats consecutifs egaux est "+m)
};;
```

Sommaire

On peut définir récursivement le flocon de Von Koch.

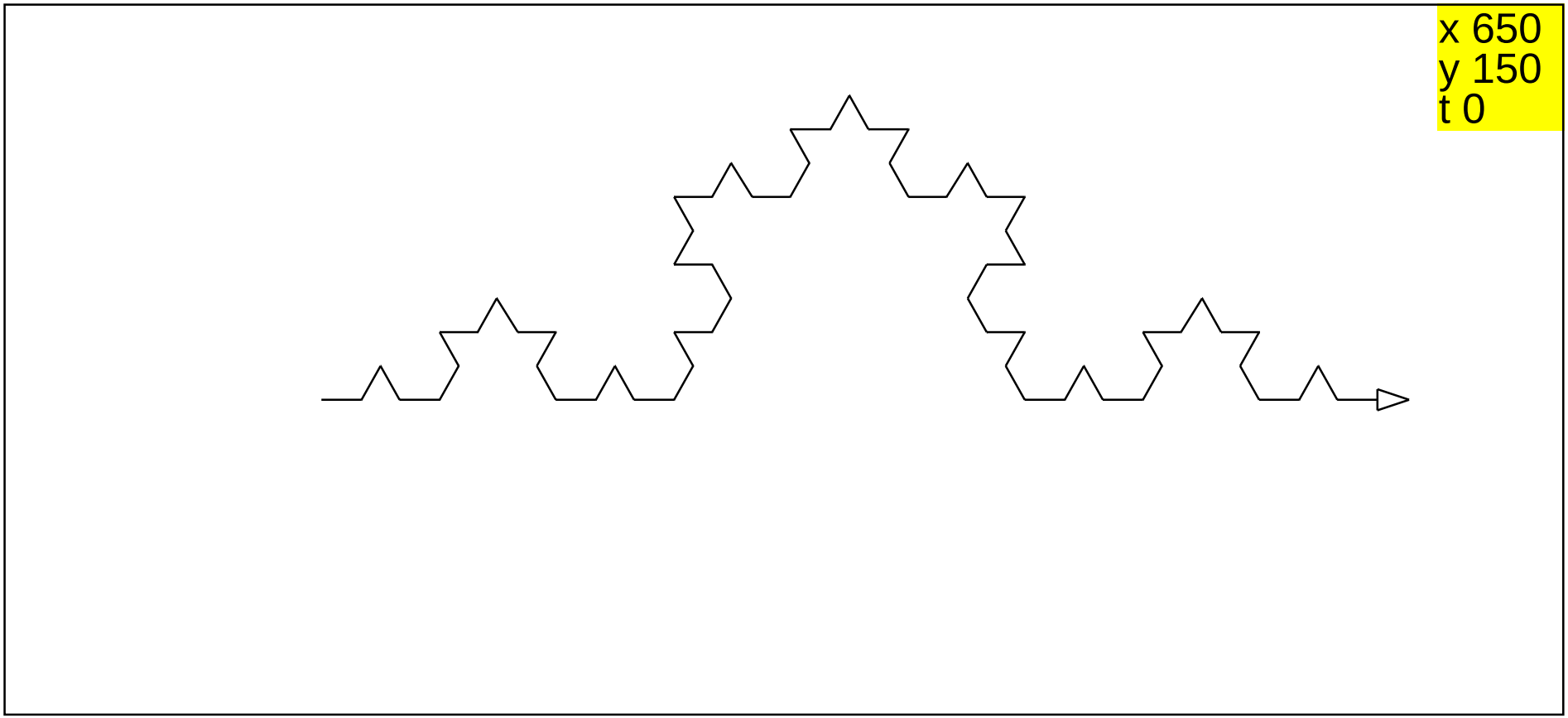
Pour simplifier, on se contentera de « floconiser » un segment.

On peut définir récursivement le flocon de Von Koch.
Pour simplifier, on se contentera de « floconiser » un segment.

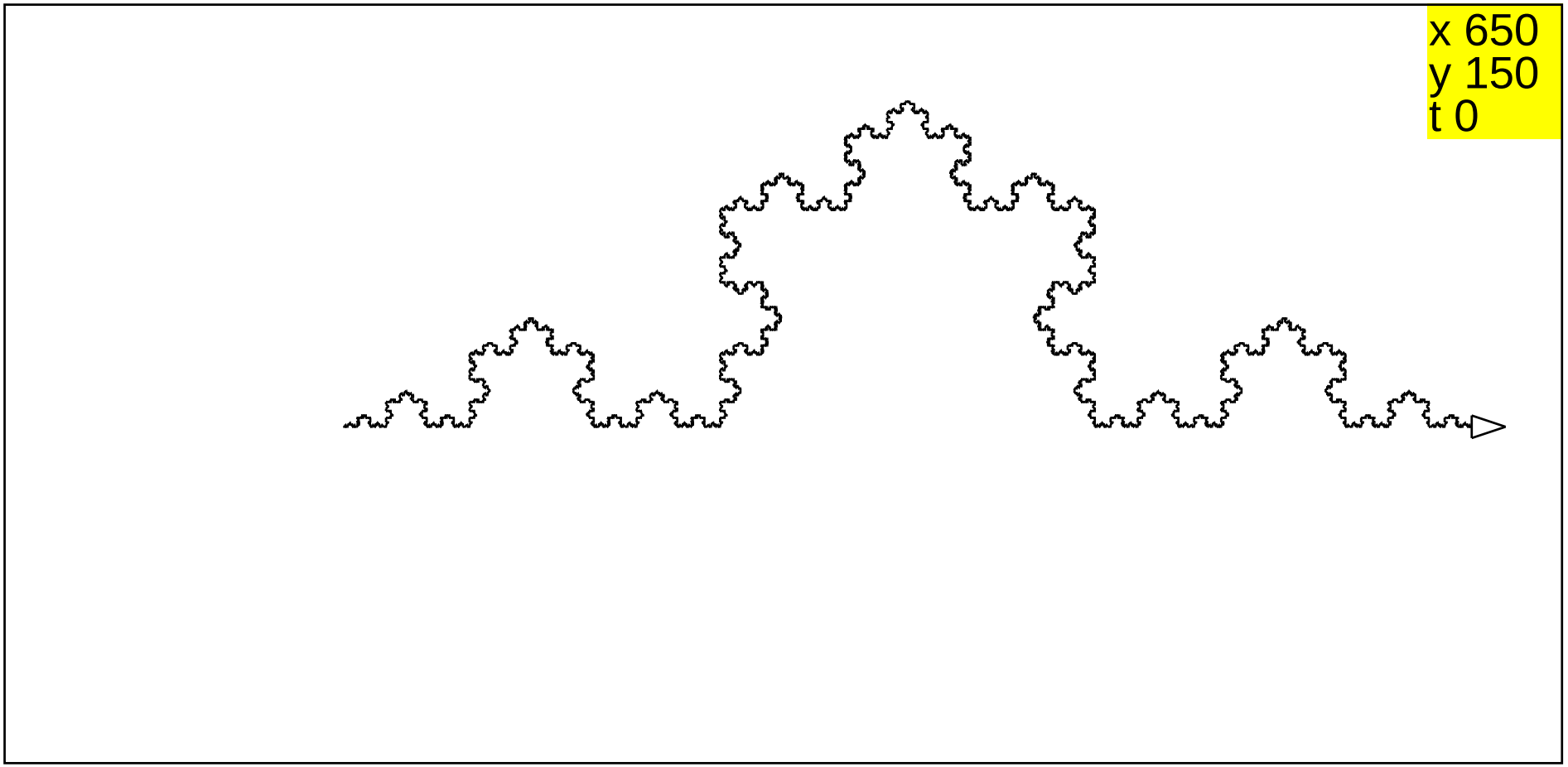
Avec la tortue

```
koch(long,n):={
si (n==0) alors
avance(long);
sinon
koch(long/3,n-1);tourne_gauche(60);
koch(long/3,n-1);tourne_droite(120);
koch(long/3,n-1);tourne_gauche(60);
koch(long/3,n-1);
fsi;
};;
```

koch(500,3)



koch(500,6)



Avec les outils géométriques de XCAS

```
KochBis(n,A,B):={
 if(n==0){segment(A,B)}
 else{
   KochBis(n-1,A,homothetie(A,1/3,B)),
   KochBis(n-1,homothetie(A,1/3,B),rotation(
       homothetie(A,1/3,B),pi/3,homothetie(A,2/3,B)
       )),
   KochBis(n-1,rotation(homothetie(A,1/3,B),pi/3,
       homothetie(A,2/3,B)),homothetie(A,2/3,B)),
   KochBis(n-1,homothetie(A,2/3,B),B)
 }
}:;
```

Flocon

```
KochBis(3,point(0,0),point(10,0)),KochBis(3,point
    (10,0),point(5,-10*sqrt(3.0)/2)),KochBis(3,
    point(5,-10*sqrt(3.0)/2),point(0,0))
```

Flocon