

# LOGO

## Géométrie Tortue, Arithmétique, Fractales et autres algorithmes ...

Alain Bois et Jean-Louis Guillot

1. Les premiers pas ...	2
2. Simulation du comportement animal Comment simuler l'odorat d'un prédateur	14
3. Fractales	16
1. Flocon de Koch	
2. Courbe C	
3. Arbres	
4. Arithmétique et suites	21
1. Calcul du PGCD et du PPCM	
2. Recherche des nombres premiers	
3. Les nombres parfaits	
4. Les congruences: Calcul de la clé RIB d'un compte chèque	
5. Calcul des éléments de la suite de Fibonacci	
6. Égalité de Bezout et récursivité non terminale	
5. Barycentre et théorème de Guldin	30
Annexe : histoire des nombres parfaits	32

# Les premiers pas ...

1. Souvenirs ...
2. Les micro-mondes
3. Comment communiquer ?
4. Des mots et des phrases pour le faire
5. Un peu de grammaire LOGO
6. Entrer les données et contrôler les procédures

## 1. Souvenirs

Il n'y a pas si longtemps, la tortue se promenait sur le sol de la classe ...

Puis elle est venue dans l'ordinateur, petit triangle bleu dans son aquarium ...

C'est bien facile de la faire se déplacer ... elle parle presque le même langage que nous.

Elle parle en LOGO. Elle connaît on utilise les mots avancer, reculer, tourner à droite, à gauche, lever le crayon, baisser le crayon, vider l'écran ... Elle sait où elle est, où elle va, elle mémorise ce qu'on lui demande de garder, elle peut nous donner les renseignements qu'elle connaît.

Elle se comporte presque comme nous :

**AV** (avance) le nombre de pas,

**RE** (recule),

**TG** (tourne à gauche) le nombre de degrés,

**TD** (tourne à droite),

**LC** (lève crayon),

**BC** (baisse crayon),

**VE** (vide écran),

**MT** (montre tortue),

**CT** (cache tortue),

**POUR** (apprends ... jusqu'au mot **FIN**)

**ME** (mixte écran) fixe le nombres de lignes de texte, sous l'espace tortue.

...

## 2. Les micro-mondes

La tortue et sa géométrie n'est pas seule dans le monde LOGO.

Elle peut vivre avec d'autres robots différents (autres tortues, chariots élévateurs, bras articulé, tables traçantes, tours, ...).

Il y a aussi le monde des nombres, le monde des mots et des listes, celui de la musique, ... et bien d'autres mondes aussi.

En bref, tous ceux que vous aurez envie d'inventer, des micro-mondes, à l'intérieur et à l'extérieur de l'ordinateur. En parlant LOGO.

### 3. Comment communiquer ?

Vous avez deux possibilités, lorsque vous travaillez avec l'ordinateur et la console jLogo (on retrouve ces principes dans la plupart des environnements LOGO) :

- **a). Pour un faire petit essai**, une mise au point, lorsque vous avez une phrase courte à tester, vous utilisez **le mode direct** (le mode par défaut).

Vous écrivez votre texte dans le "champ de saisie" situé à côté du symbole d'invite de LOGO, le point d'interrogation ("?"). LOGO est à votre écoute.

Dès le retour de chariot, à la fin de votre phrase, LOGO interprète ce que vous avez écrit et renvoie un résultat, exécute une action (ou renvoie le message d'erreur approprié).

Pour commander le calcul "écris le reste de la division de 119 par 27", vous tapez :

```
EC RESTE 119 27
```

Pour obtenir une ligne polygonale :

```
REPETE 25 [AV 10 TD 5]
```

Lorsqu'on est en mode direct, LOGO garde la trace de toutes les commandes qu'on a écrites. On peut utiliser le "copier-coller" pour mettre au point la phrase travaillée.

- **b). Pour écrire un texte plus long** (un paragraphe avec plusieurs phrases - les lignes d'instructions - ou une phrase complexe), vous passez par le **mode Éditeur**. La validation sera possible en sortant de l'Éditeur.

Pour entrer dans l'Éditeur tapez ED lorsque vous êtes en mode direct, ou utilisez l'un des menus disponibles.

**ED** permet de retrouver la fenêtre d'édition. Elle est vide si c'est le premier passage, sinon on retrouve la page précédente et on l'utilise comme un brouillon, avec les fonctions d'édition habituelles.

**ED [ ]** permet de passer dans l'éditeur en le vidant.

**ED [CARRE]** permet de passer dans l'éditeur en le vidant, et ramène le texte de la procédure CARRE, si elle a été bien définie avec **POUR** et **FIN** (voir ci - dessous).

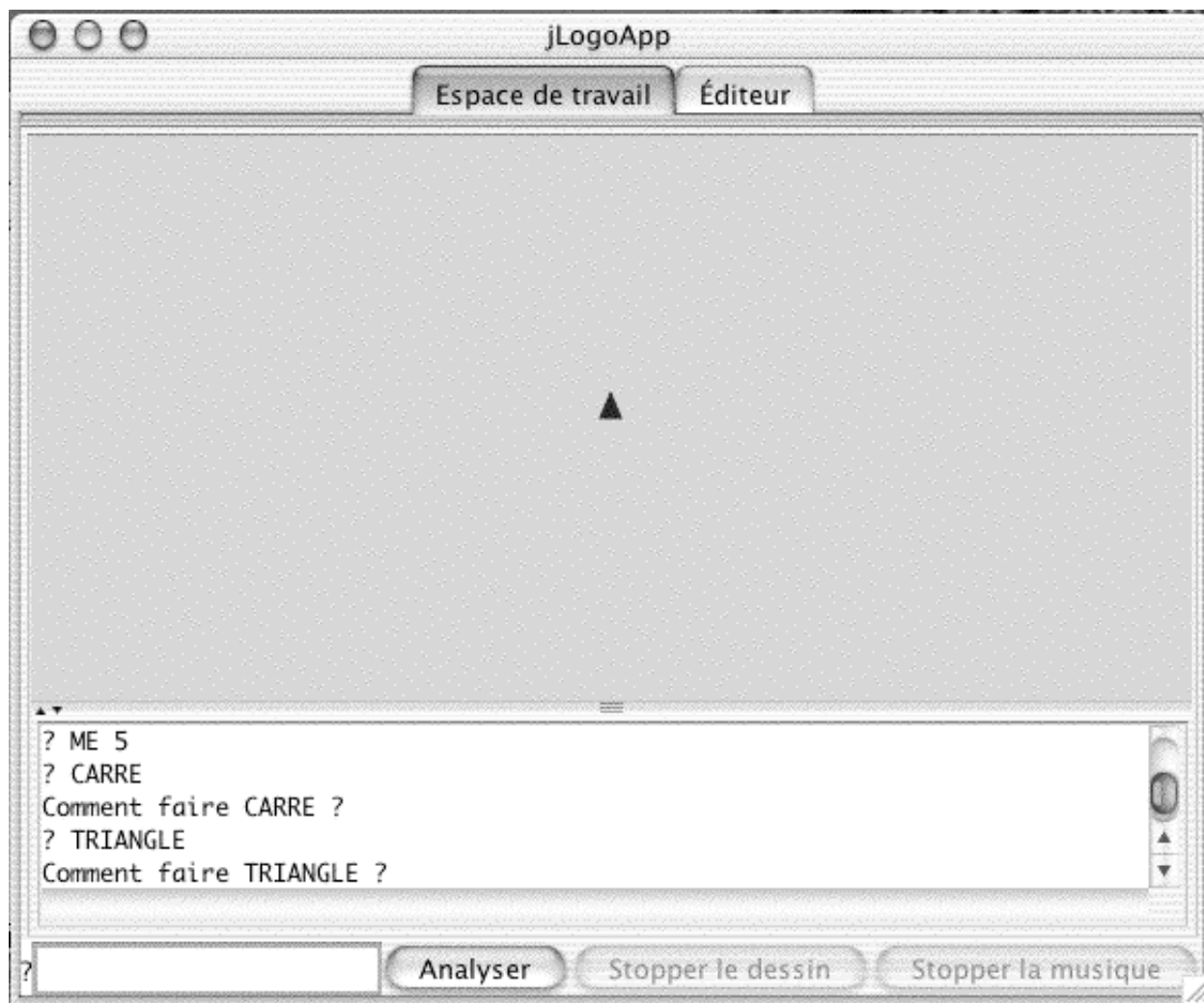
Pour mettre au point une phrase ("une ligne d'instructions") comme la suivante :

```
AV 100 TD 90 AV 100 TD 90 AV 50 TD 90 AV 50 TD 90 AV 100 TD 90 AV  
25 TD 90 AV 25 TD 90 AV 50
```

il est vivement recommandé d'utiliser l'éditeur ! Les erreurs de frappe sont souvent nombreuses !



Avec jLogo, sur certains environnements, vous pourrez utiliser les onglets :



Vous pourrez dans ce cas aussi diminuer la partie graphique en relevant la barre qui la sépare du texte du "journal" en utilisant la souris.

Vous pourrez également toujours utiliser la procédure ME.

ME 5

réservera 5 lignes de texte pour afficher le "journal" de vos commandes et des réponses jLogo.

Dans l'exemple ci dessus, CARRE et TRIANGLE (équilatéral) ne figurent pas parmi les primitives ... vous devrez donc les apprendre à LOGO, comme vous avez fait pour TRUC.

## 4. Un peu de grammaire LOGO

### - a). Les mots et les listes

Nous décrivons ici seulement les éléments qui vont nous servir dans la suite. De nombreux objets LOGO sont disponibles ou peuvent être créés.

Les objets LOGO pour lire et écrire sont **les mots** et **les listes**. Les mots sont écrits avec les **caractères** usuels et séparés par les espaces. **Les nombres** sont des mots particuliers ne comportant que des chiffres et s'il y a lieu la virgule ou le point pour séparer la partie décimale.

Les noms des primitives et des procédures (les "exécutables") sont des mots comme AV, REPETE, TRUC ...

Si **un mot** doit être pris comme une constante non exécutable, comme une donnée ou un identificateur (nom de variable) on le fait **précéder d'un guillemet double** sauf si c'est un nombre et sauf s'il est un élément d'une liste.

"CANARD	<- un mot
125	<- un mot qui est un nombre entier
1357,89	<- un mot qui est un nombre décimal
[CANARD 125 1357,89]	<- la liste des trois mots ci-dessus
[maison "5" CANARD "125 [1 a 3]]	<- une liste de trois éléments

**Une liste** est une suite de mots ou de listes.

Vous devrez l'écrire **entre crochets**, mais LOGO vous la reproduira en supprimant les crochets de début et de fin.

EC [maison "5" CANARD "125 [1 a 3]]	<- vous demandez à LOGO d'écrire
maison "5" CANARD "125 [ 1 a 3 ]	<- voici la réponse de LOGO

EC "CANARD	<- vous demandez à LOGO d'écrire
CANARD	<- voici la réponse de LOGO

### - b). Les opérations ou fonctions et les commandes

**Les opérations (ou fonctions) produisent un résultat (un objet utilisable) et les commandes réalisent une action sur l'environnement.**

EC 2 + 3	<- opération addition "infixée"
EC SOMME 2 3	<- opération addition "préfixée"

Les deux **instructions** produisent 5 comme résultat. La **commande** EC écrit le nombre 5. Si vous oubliez de taper EC, LOGO dira :

Que faire de 5 ?

C'est comme ça que vous ferez des phrases compliquées.

AV SOMME PROD 2 10 RESTE 24 5	
AV SOMME PROD RESTE 24 5 2 10	c'est différent !

Vous pouvez construire des procédures qui sont des commandes, d'autres qui

sont des opérations (ou fonctions), d'autres qui sont les deux à la fois.

Certaines fonctions n'ont **aucune entrée** (argument, paramètre) :

POSITION rend la liste des coordonnées de la tortue  
CAP rend le cap de la tortue

Certaines fonctions ont **une seule entrée** :

PREM liste rend le premier élément de la liste ou le premier caractère du mot  
RC nombre rend la racine carrée du nombre

Certaines fonctions ont **deux entrées** :

MOT "car "table rend le mot "cartable

Si on dispose de **plusieurs objets en entrées**, on peut s'organiser (ne tapez pas le point d'interrogation):

? EC PH PH PH PH MOT MOT "4 "+" "3 [est la somme de] "4 "et "3  
4+3 est la somme de 4 et 3

En plus des fonctions mathématiques, et de celles qui opèrent avec les mots et les listes et qui rendent des nombres, des mots ou des listes, LOGO fournit des **prédicats**.

Un **prédicat** est une fonction comme celles que nous venons de décrire, mais elle rend comme résultat le mot "VRAI ou le mot "FAUX. Les opérations booléennes sont des prédicats qui ont aussi comme entrée les valeurs booléennes "VRAI ou "FAUX.

C'est très commode à utiliser pour écrire les conditions des tests.

EC ET PLG? RC 2 1,41 PLP? RC 2 1,42  
VRAI

Il est bien vrai que  $\sqrt{2}$  est plus grand que 1,41 et plus petit que 1,42

ET est une opération booléenne préfixée ayant deux entrées booléennes.  
PLG? et PLP? sont deux prédicats ayant chacun deux nombres comme entrée.

-c) Comment faire une nouvelle fonction ?

C'est facile à comprendre lorsque vous n'avez pas d'entrée (pas de paramètres) à entrer par exemple, si vous voulez obtenir l'abscisse de la position de la tortue vous définirez la fonction XCOR. Le mot magique est RENDS qui produit un objet directement utilisable dans vos phrases comme EC SOMME 15 XCOR.

POUR XCOR  
REND S PREMIER POSITION  
FIN

L'entrée des données sera détaillée au paragraphe suivant. Vous pourrez alors créer avec le mot magique RENDS la procédure OPPOSE, avec une entrée, inconnue en LOGO pour l'utiliser comme SOMME, DIFF, DIV, QUOT, PROD, RESTE ....

## 4. Entrer les données et contrôler les procédures

Étudions maintenant l'affectation des variables, les tests, la répétition, la récursivité et la récursivité non terminale.

### - a). Comment passer les données ?

Il y a bien des primitives comme LISCAR (qui prend le caractère tapé au clavier) ou LISLISTE (qui prend toute la phrase tapée au clavier jusqu'au retour de chariot), qui ressemblent un peu au get ou à l'input ou bien au readln ou read que certains connaissent. De toute façon, il faut bien ranger quelque part une donnée qui vient du clavier, ou l'utiliser tout de suite dans une fonction ou une commande pour en faire quelque chose.

La plupart du temps, en LOGO, vous êtes directement dans l'espace de travail, au commandes de tout l'environnement. Vous allez tout de suite agir avec les primitives ou les procédures qui vous sont nécessaires, et vous leur donnez directement les données (arguments ?) dont elles ont besoin. Admettons par exemple que vous ayez besoin de faire des carrés de différentes tailles. Vous écrirez la procédure :

```
POUR CARRE :COTE  
REPETE 4 [AV :COTE TD 90]  
FIN
```

Puis une fois qu'elle sera validée, vous commanderez en mode direct, suivant ce qu'il vous plaira :

```
CARRE 10  
CARRE 50  
etc...
```

Tout se passe comme vous procédiez chaque fois à une **affectation de variable**, mais c'est fait directement à l'appel de la procédure (sans "input", "read" ...).

En fait, vous avez créé la variable dont le **nom est le mot "COTE** et dont le **contenu est désigné par :COTE** (valant successivement 10, puis 50, puis ...). Cette variable a été **créé dans le titre de la procédure**; c'est **une entrée** de la procédure CARRE.

Vous pouvez l'**utiliser dans le corps de la procédure** comme bon vous semble. Par exemple, insérez l'instruction suivante, puis testez la nouvelle procédure.

```
EC PH PH [Je viens de faire le carré de] :COTE [pas de côté.]
```

Une variable peut également être définie dans le corps d'une procédure ou directement au niveau supérieur, sans avoir lancé de procédures.

On utilise les commandes **DONNE** ou **SOIT**. Tapez en mode direct les deux commandes et vous retrouvez dans votre tirelire ce que vous y aurez mis.

```
? DONNE "tirelire "20  
? EC :tirelire  
20
```

De façon analogue essayez :

```
? SOIT "X 12,5
```

? EC :X  
12,5

Le contenu de la tirelire est le mot 20 et celui de X est le nombre 12,5

Contrairement à l'apparence, ces deux affectations de variables fonctionnent différemment .

La commande "DONNE produit une **variable globale**, c'est à dire connue de toutes les procédures qui l'appelleront ou de vous-même, au niveau supérieur.

Par contre, la commande "SOIT produit une **variable locale**, qui n'est connue que par les niveaux d'appels inférieurs à celui où elle a été créée.

Comme vous avez fait "X au niveau supérieur, tout le monde verra son contenu, comme si c'était une variable globale. Par contre la variable "COTE créée par la procédure CARRE n'est pas accessible au niveau supérieur, c'est une variable locale. Tout se passe comme si vous l'aviez définie en utilisant la commande SOIT à l'intérieur de la procédure CARRE.

? EC :COTE  
Pas de chose donnée à COTE

### - b). Les répétitions et les tests.

Comme structure itérative, LOGO utilise le mot **REPETE** et la **récurtivité**.

#### Répétition

REPETE le nombre de fois indiqué la liste d'instructions qui est écrite.

#### Récurtivité

Une procédure récursive est une procédure qui s'appelle elle même. Pensez plutôt que des procédures "clones" s'appellent successivement :

La première appelle la deuxième qui appelle la troisième qui appelle ... en espérant qu'un événement arrêtera cette chaîne infernale.

```
POUR CARRE :COTE  
REPETE 4 [AV :COTE TD 90]  
CARRE :COTE + 10  
FIN
```

Plutôt que d'attendre que la tortue se cogne contre les murs, ou que l'ordinateur fonde, vous préférerez prévoir un **test d'arrêt** lorsque vous entreprendrez une récurtivité.

### **Condition simple si ... alors :**

```
si condition
    alors action
suite
```

```
POUR CARRE :COTE
REPETE 4 [AV :COTE TD 90]
SI PLP? :COTE 100 [CARRE :COTE + 10]
FIN
```

N'oubliez pas de mettre sur la même ligne d'instructions le SI, la condition (prédicat) et la liste d'instructions (entre crochets). Lorsque la condition n'est pas réalisée, on passe à la suite, (ligne du dessous) qui peut être le mot FIN

Dans l'exemple ci dessus vous avez en fait reconstitué une structure du type

```
faire action
    jusqu'à condition
suite
```

En changeant la place du test, il est facile de réaliser la structure

```
tant que condition
    faire action
suite
```

Vous reconstituez de la sorte les structures itératives élémentaires des autres langages de programmation avec récursivité et condition simple.

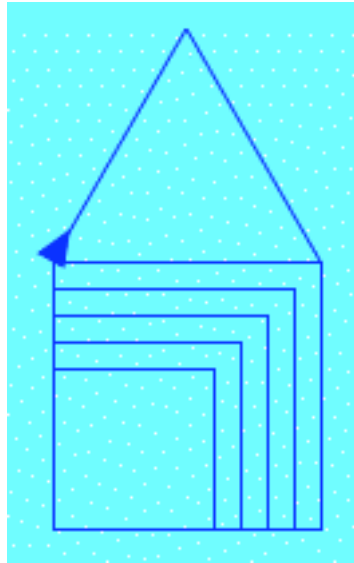
### **Condition alternative si ... alors .. sinon :**

```
si condition
    alors action 1
    sinon action 2
suite
```

```
POUR CARRE :COTE
REPETE 4 [AV :COTE TD 90]
SI PLP? :COTE 100 [CARRE :COTE + 10][AV :COTE TD 30 REPETE 3 [AV
:COTE TD 120]]
FIN
```

Comme précédemment, n'oubliez pas de mettre sur la même ligne d'instructions le SI, la condition (prédicat), et la liste d'instructions (entre crochets) à faire lorsque la condition est vraie, puis la liste d'instructions (entre crochets) à faire lorsque la condition est fausse. Après l'exécution de l'une de ces deux listes, on passe à la suite.

Si vous tapez CARRE 60 vous obtiendrez la figure suivante.



Imaginez bien les procédures qui s'appellent successivement (ce sont les acteurs, les ouvriers qui s'activent dès que vous avez passé votre commande):

```
CARRE 60
  action 1 (dessin du carré de 60)
  test VRAI
    CARRE 50
      action 1 (dessin du carré de 60)
      test VRAI
        ....
        CARRE 100
          action 1 (dessin du carré de 60)
          test FAUX
            action 2 (déplacement puis triangle)
            FIN
```

Le travail est fini. Le travail semble fini ...  
Regardons d'un peu plus près.

### **Récurtivité non terminale :**

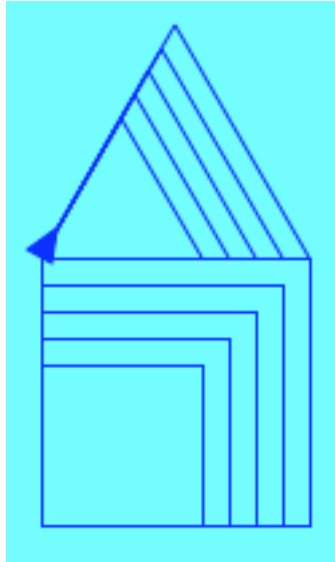
Modifiez la procédure précédente pour obtenir :

```
POUR CARRE :COTE
  REPETE 4 [AV :COTE TD 90]
  SI PLP? :COTE 100 [CARRE :COTE + 10][AV :COTE TD 30]
  REPETE 3 [AV :COTE TD 120]
  FIN
```

Essayez votre nouvelle procédure :

CARRE 60

Vous obtenez ceci :



Étonnant, non ? ... et ce n'est pas une erreur de LOGO !

En effet, lorsque appels successifs sont terminés, le dernier ouvrier qui intervient, CARRE 100 exécute dans son test AV 100 TD 30, puis passe à la ligne suivante pour faire son triangle de 100.

Mais il n'a pas fini contrairement à ce que vous auriez pu croire ...

Par contre, il dit à CARRE 90 qui lui avait passé la commande que son travail est fini. Mais CARRE 90 remarque qu'il n'avait pas fini sa propre liste de commandes : il avait encore une instruction à faire après avoir fait travaillé son collègue. Il fait donc son triangle de 90 ... ensuite, il doit "rendre la main" à celui qui l'avait commandé lui aussi a des comptes à rendre ! Et ils se repassent la main... Jusqu'à ce que CARRE 60 termine sa liste et vous rende la main à vous le véritable chef : tous les travailleurs ont terminé votre commande.

Pensez aussi à tous ces documents que vous empilez; pour revenir au premier, il faut bien refaire tout le chemin à l'envers, tout dépiler. Ici on dit souvent descendre la récursivité, puis la remonter.

**- d). Faites des fonctions avec la récursivité :**

Sommez une liste : et faites la moyenne d'une liste de nombres :

```
POUR TOTAL :L
SI VIDE? :L [RENDS 0] [RENDS SOMME PREM :L TOTAL SP :L]
FIN
```

Faites la moyenne d'une liste de nombres :

```
POUR MOYENNE :L
RENDS DIV TOTAL :L COMPTE :L
FIN
```

Attention : DIV donne le quotient alors que QUOT donne le quotient entier.

```
? EC MOYENNE [12 16 18 14,5 13]
14,7
```

Mettez les mots à l'envers :

```
POUR ENVERS :M
SI VIDE? :M [RENDS "[RENDSD MOT DER :M ENVERS SD :M]
FIN
```

DER pour dernier, et SD pour sauf dernier, agissent ici sur mots et caractères.

```
? ENVERS "LAPIN
Que faire de NIPAL ?
```

# Simulation du comportement animal

## Comment simuler l'odorat d'un prédateur ?

Nous allons situer sur l'écran une proie dont la position va être définie par l'utilisateur et l'emplacement initial du prédateur (dans notre cas la tortue écran). Pour cela nous écrivons une procédure d'initialisation comme suit:

```
POUR INITIALISATION
EC [TAPEZ LA POSITION EN X ET Y DU PREDATEUR]
DONNE "POSPREDATEUR LL
EC [TAPEZ LA POSITION EN X ET Y DE LA NOURRITURE]
DONNE "POSNOURRITURE LL
LC FPOS :POSNOURRITURE
BC CROIX
LC FPOS :POSPREDATEUR
DONNE "ANGLE_ALEATOIRE HASARD 60
FIN
```

```
POUR CROIX
AV 2 RE 4 AV 2 TD 90 AV 2 RE 4
FIN
```

Ça va c'est pas trop dur!!!

Maintenant venons à la simulation simple de l'odorat, on considère que la tortue est sur le bon chemin lorsque la distance à la proie se raccourcit — dans ce cas elle continue à avancer — sinon on décide aléatoirement de la faire tourner de 1 degré vers la droite.

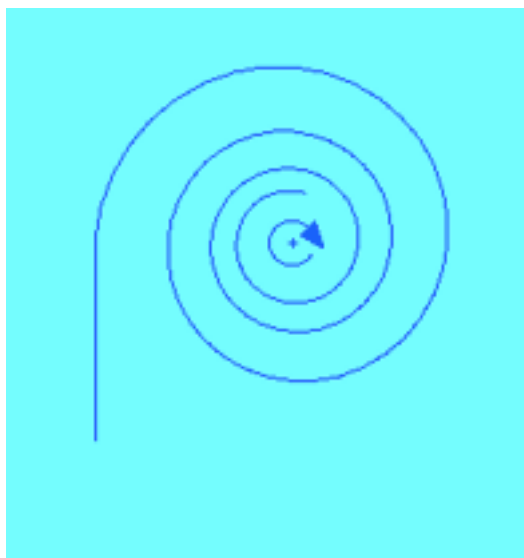
On va écrire une procédure-prédicat qui va signaler à la tortue que son odorat faiblit:

```
POUR SENTE_MOINS_BONNE?
RENDS PLG? DISTANCE POS :POS_NOURRITURE :DERNIERE_DISTANCE
FIN
```

Il nous faut initialiser la variable :DERNIERE\_DISTANCE dans la procédure qui va lancer la recherche de nourriture.

```
POUR CHERCHER_NOURRITURE
DONNE "DERNIERE_DISTANCE DISTANCE POS :POS_NOURRITURE
AV 1
SI SENTE_MOINS_BONNE? [TD :ANGLE_ALEATOIRE]
TROUVER_NOURRITURE
FIN
```

Il nous reste à tester cette procédure pour voir l'effet produit :



Voici deux copies d'écran avec un angle de 10 degrés pour la première et un angle de 20 degrés pour la deuxième.

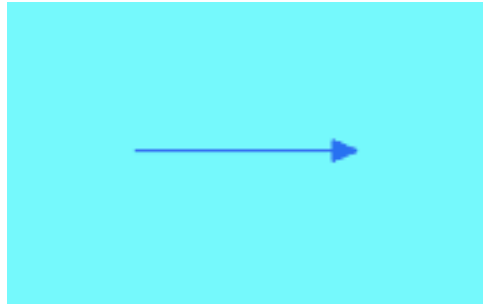
Il est possible de prévoir de l'aléatoire dans les marches de la tortue quand elle se rapproche de la proie ou jouer sur de l'aléatoire dans les angles de rotation.

# Fractales

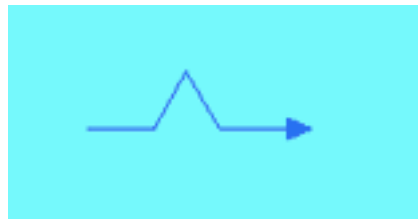
1. Flocon de Koch
2. Courbe C
3. Arbres

## 1. Flocon de Koch

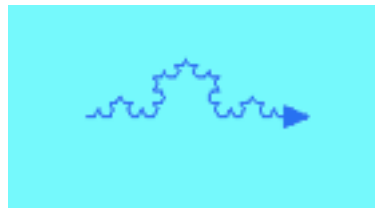
C'est quoi au juste un flocon de Koch. Ça dépend du niveau de récursivité !! Au niveau 0, c'est juste un triangle équilatéral dont les côtés comme tous les côtés sont des simples traits.



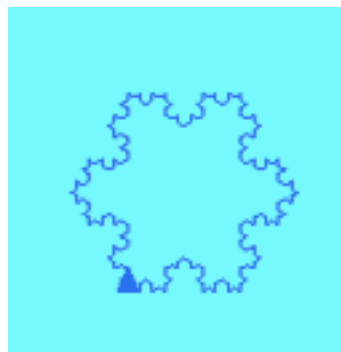
Puis les traits se divisent eux-mêmes en 4 traits comme ceci:



Et ainsi de suite:



Et donc un flocon devient l'assemblage de ces nouveaux traits :

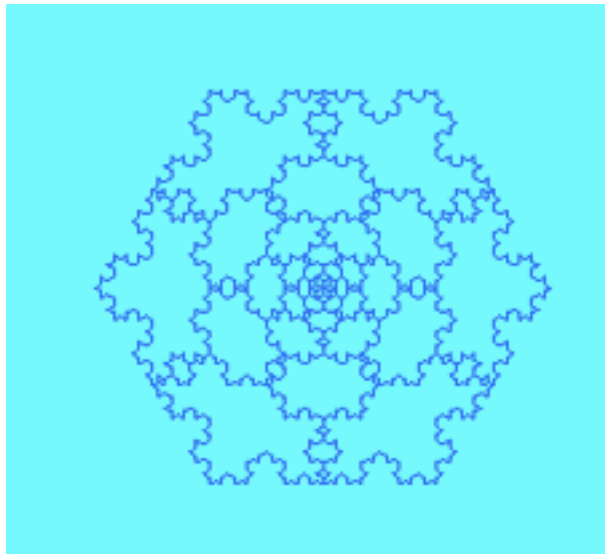


Voici comment on programme en Logo tout cela:

```
POUR COTE :TAILLE :NIVEAU
  SI EGAL? :NIVEAU 0 [AV :TAILLE STOP]
    COTE :TAILLE / 3 DIFF :NIVEAU 1
    TG 60
    COTE :TAILLE / 3 DIFF :NIVEAU 1
    TD 120
    COTE :TAILLE / 3 DIFF :NIVEAU 1
    TG 60
    COTE :TAILLE / 3 DIFF :NIVEAU 1
  FIN
```

Expliquons un peu: si le niveau de récursivité est 0, la procédure va s'arrêter grâce au STOP après avoir tracé un trait simple "AV :TAILLE" mais si le niveau de récursivité est seulement 1, ça se complique: on croit naïvement quand on ne s'est pas attaqué à une récursivité non terminale que la procédure retourne sur elle-même et qu'elle va donc faire dessiner à l'objet "Tortue" un trait 3 fois plus petit "COTE :TAILLE / 3" et que la procédure va s'arrêter. Et non!!! elle va continuer sur TG 60 et COTE :TAILLE / 3 DIFF :NIVEAU 1 donc faire le deuxième côté mais TAILLE et NIVEAU sont les variables de la procédure appelante donc ne changent pas de valeurs et ainsi de suite jusqu'à ce que le mot FIN de la procédure appelante soit détecté. Oh les belles cascades !!!

Pour le plaisir des yeux voici ce qu'on obtient si on tape tout bêtement:  
REPETE 6 [FLOCON 100 3 TD 60]



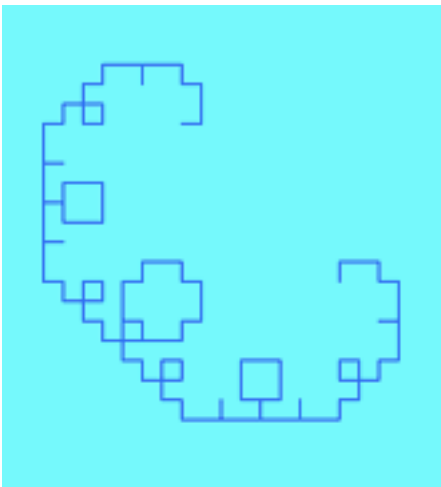
Avouez que c'est pas mal !!!!

L'IREM de Nantes a publié une brochure en 1990: "Les courbes de Von Koch et de Sierpinski comme paradigmes de fractals" par A. Robert, professeur à l'université de Neufchâtel. Il y a des exercices sur les centres de gravité ou autres qui peuvent être traités élégamment avec l'aide de Logo.

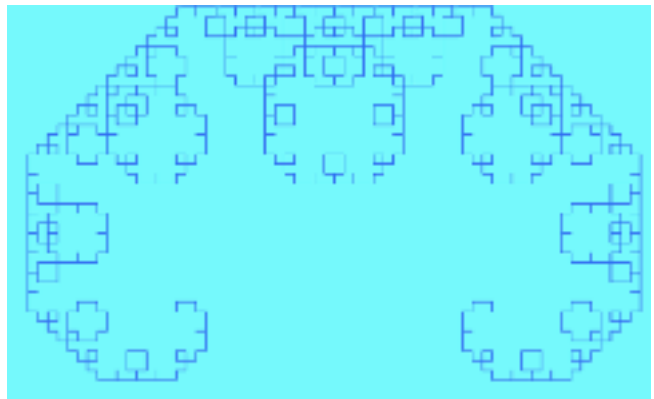
## 2. Courbe "C"

Sans commentaires mais avec les procédures et les images-écrans, voici la courbe appelée C.

```
POUR C :TAILLE :NIVEAU
SI :NIVEAU = 0 [AV :TAILLE STOP]
C :TAILLE DIFF :NIVEAU 1
TD 90
C :TAILLE :NIVEAU - 1
TG 90
FIN
```



C 10 7

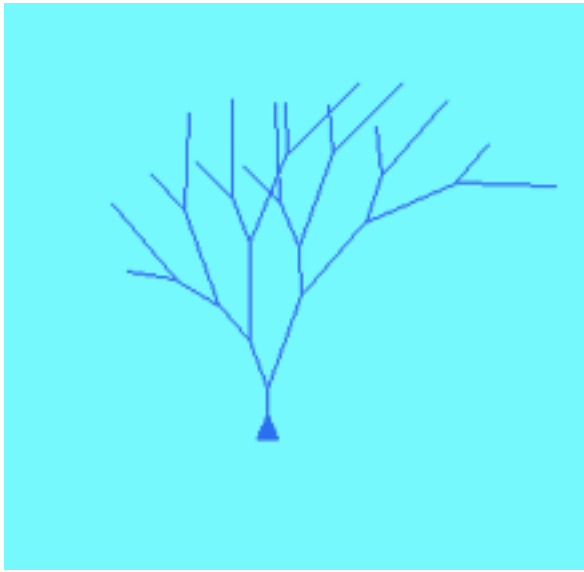


C 10 10

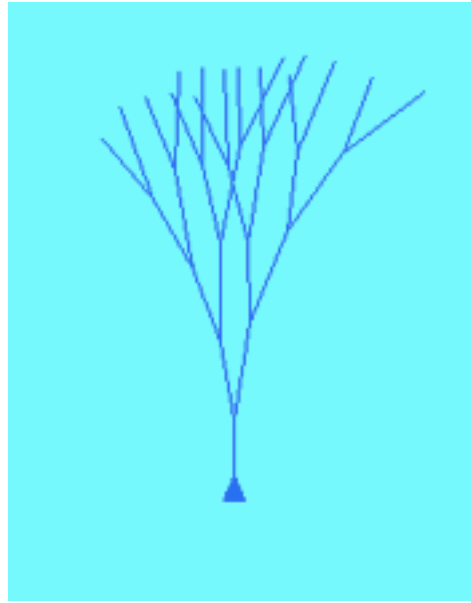
Vous voyez pourquoi on l'appelle courbe "C". Selon le niveau de la récursivité il y a des études intéressantes à mener sur l'emplacement de l'axe de symétrie.

Nous terminerons cette étude par la modélisation d'un arbre sur lequel on n'a pas placé des fruits mais ça pourrait être un bon exercice à faire à la maison !!!

### 3. Arbres



BRANCHEG 50 20 4 0.5



BRANCHEG 50 10 4 0.8

Et le bonsaï



BRANCHEG 10 10 10 0.8

Rien ne vous empêche de dessiner une forêt ou des arbres couchés par le vent etc...

Allez on vous donne les procédures, à vous de les tester, de les modifier.

```
POUR BRANCHEG :LON :ANG :NIVEAU :COEF  
  AV :LON * :COEF  
NOEUD :LON :ANG :NIVEAU :COEF  
  RE :LON * :COEF  
FIN
```

```
POUR NOEUD :LON :ANG :NIVEAU :COEF
  SI EGAL? :NIVEAU 0 [STOP]
    TG :ANG
    BRANCHEG :LON :ANG DIFF :NIVEAU 1 :COEF
    TD 2 * :ANG
    BRANCHED :LON :ANG + 2 DIFF :NIVEAU 1 :COEF
    TG :ANG
```

FIN

```
POUR BRANCHED :LON :ANG :NIVEAU :COEF
  AV :LON
  NOEUD :LON :ANG :NIVEAU :COEF
  RE :LON
```

FIN

Vous commencez par BRANCHEG ou BRANCHED ça fera la même chose.

# ARITHMÉTIQUE et suites

1. Calcul du PGCD et du PPCM
2. Recherche des nombres premiers
3. Les nombres parfaits
4. Les congruences: Calcul de la clé RIB d'un compte chèque
5. Calcul des éléments de la suite de Fibonacci
6. Égalité de Bezout et récursivité non terminale

## 1. Calcul du PGCD et du PPCM

Nous allons utiliser l'algorithme d'Euclide :

```
Tant que a ≠ b
  faire
    si a > b
      a ← a - b
      b ← a
    sinon
      a ← b - a
      b ← a
  fin si
fin Tant que
```

Ce qui donne en Logo:

```
POUR PGCD :A :B
  SI EGAL? :A :B [RENDS :A ]
  SI PLG? :A :B [RENDS PGCD :B :A - :B]
  RENDS PGCD :B - :A :A
FIN
```

Si vous voulez obtenir le PGCD par la méthode des divisions successives, vous pourrez vous contenter d'écrire :

```
POUR PGCD :A :B
  SI EGAL? 0 RESTE :A :B [RENDS :B ][RENDS PGCD :B RESTE :A :B]
FIN
```

... ce qui dit exactement : pour avoir le PGCD de A et de B, si B divise A, c'est B, sinon, ce sera le PGCD de B et du reste de la division précédente. Vous êtes en train de faire les fameuses divisions en cascade ... élégant ? non ?

Vous pouvez tester ces procédures et naturellement écrire des "choses" comme cela :

```
EC PGCD 24 36 mais aussi AV PGCD 360 144 et comme nous le verrons plus loin rien ne nous empêche à condition d'avoir défini FIBO d'écrire
PGCD FIBO 15 FIB 30 ou FIBO PGCD 15 30
```

Il est donc possible d'écrire une procédure pour calculer le PPCM de 2 nombres de la façon suivante :

```

POUR PPCM :A :B
  RENDS QUOT PROD :A :B PGCD :A :B
FIN

```

Pendant qu'on y est pourquoi s'arrêter au PPCM de de deux nombres, on peut écrire le PPCM de n nombres pris dans une liste de la façon suivante :

```

POUR PPCMS :LISTE
  SI EGAL? COMPTE :LISTE 2 [REND S PPCM PREM :LISTE DER :LISTE]
  RENDS PPCMS PH PPCM ITEM 1 :LISTE ITEM 2 LISTE SP SP :LISTE
FIN

```

Essayez PPCMS [24 36 48] par exemple ou encore PPCMS [15 25 36 42].

**Autre idée** bien commode, (utilisée parfois pour trouver un dénominateur commun à deux fractions) : **vous prenez le plus grand des deux nombres, vous regarder si c'est un multiple du petit**, sinon vous prenez son double, vous regardez ... puis les multiples successifs jusqu'à ce que vous ayez un multiple commun.

Ce qui donne en LOGO  
Tant que vous n'avez pas ce multiple, vous faites tourner la moulinette PPMC1 :

```

POUR PPMC :A :B
  SOIT "N 1
  RENDS PPMC1 :A :B :N
FIN

POUR PPMC1 :A :B :N
  SI EGAL? 0 RESTE :N * :A :B [REND S :N * :A]
  RENDS PPMC1 :A :B :N + 1
FIN

```

## 2. Recherche des nombres premiers

- a) Si on y va par la **méthode forte**, on peut écrire tout d'abord une procédure nous permettant d'**écrire la liste des diviseurs** d'un nombre, cela nous servira par la suite pour tester les nombres parfaits.

```

POUR LISTE_DIVISEURS :NOMBRE
  SI EGAL? :NOMBRE 1 [REND S 1]
  RENDS DIVISEURS :NOMBRE 1
FIN

POUR DIVISEURS :NOMBRE :DEBUT
  SI EGAL? :DEBUT RC :NOMBRE [REND S :DEBUT]
  SI PLG? :DEBUT RC :NOMBRE [REND S "]
  SI EGAL? RESTE :NOMBRE :DEBUT 0 [REND S PH :DEBUT PH DIVISEURS
:NOMBRE :DEBUT + 1 :NOMBRE / :DEBUT]
  RENDS DIVISEURS :NOMBRE :DEBUT + 1
FIN

```

### Deux remarques :

Il faut s'habituer à l'écriture préfixée des prédicats SI EGAL? ou SI PLG?  
l'écriture de la récursivité est originale :

```
RENDS PH :DEBUT PH DIVISEURS :NOMBRE :DEBUT + 1 :NOMBRE / :DEBUT
```

On remplit la procédure-objet par les deux bouts : au début le diviseur trouvé, à la fin le nombre divisé par ce diviseur et au milieu la suite des autres diviseurs.

Et donc pour savoir si un nombre est premier, il suffit d'écrire la procédure suivante.

```
POUR PREMIER? :A  
RENDS EGAL? COMPTE LISTE_DIVISEURS :A 2  
FIN
```

Ça marche mais c'est un peu lourd. Voyons quelque chose de plus élégant.

En calquant notre démarche sur le crible d'Ératosthène, voici ce que nous pourrions écrire en LOGO :

Commençons par le plus simple, savoir si un nombre est premier en donnant comme liste de début [2 3]. Cette procédure ne marche que pour les nombres supérieurs à 3.

Comme vous avez pu déjà le voir, il y a aussi en Logo des procédures-objets - prédicats qui rendent une valeur de vérité.

```
POUR PREMIER? :A :B  
SI VIDE? :B [RENDS "VRAI]  
SI EGAL? RESTE :A PREM :B 0 [RENDS "FAUX]  
RENDS PREMIER? :A SP :B  
FIN
```

Je vous dois quelques explications :

- la procédure va tester si A est premier en se servant de la liste B qui va être initialisée à [2 3 5 7] par exemple.
- si le reste de A par le premier élément de B est nul c'est pas la peine d'aller plus loin, on a trouvé un diviseur donc on peut rendre Faux, sinon on recommence mais avec une liste de diviseurs moins longue, c'est le principe du crible. Essayez PREMIER? 20 [2 3 5] ça marche mais si vous essayez PREMIER? 49 [2 3 5], l'ordinateur vous répond VRAI, CE QUI EST FAUX!!!

Il faut donc agrandir le crible, ce que l'on va faire dans cette deuxième mouture :

```
POUR LISTE_PREMIERS :A :B  
SI PLG? DER :A :B [RENDS "]  
DONNE "NBRE SOMME DER :A 2  
SI PREMIER? :NBRE :A [RENDS PH :NBRE LISTE_PREMIERS PH :A :NBRE :B]  
RENDS LISTE_PREMIERS PH :A :NBRE :B  
FIN
```

Dès que l'on trouve un nombre premier on l'ajoute à la variable B, alors on ne divise plus par 2 et 3 seulement mais par la liste des nombres premiers qui s'allonge au fur-et-à-mesure que l'on en trouve un.

L'inconvénient c'est que 2 et 3 sont oubliés, je n'ai pas trouvé de solution plus

élégante que de les rajouter dans la procédure suivante qui va donner sous forme de liste les nombres premiers jusqu'à 200.

```
POUR CRIBLE_ERATOSTHENE
  RENDS PH [2 3] LISTE_PREMIERS [2 3] 200
FIN
```

- **b)** On peut aussi essayer la **méthode directe plus douce** : un nombre est premier lorsqu'il n'a **aucun diviseur entre 1 et sa racine carrée** (donc entre 1 et lui-même !). On regarde s'il est pair sinon on regarde s'il est divisible par les nombres impairs, jusqu'à sa racine carrée :

```
POUR PREM? :N
  SI OU EGAL? :N 0 EGAL? :N 1 [REND "FAUX]
  SI OU EGAL? :N 2 EGAL? :N 3 [REND "VRAI]
  SI EGAL? 0 RESTE :N 2 [REND "FAUX]
  RENDS NON DIV? 3
FIN
```

```
POUR DIV? :D
  SI EGAL? :D * :D :N [REND "VRAI]
  SI PLG? :D * :D :N [REND "FAUX]
  SI EGAL? RESTE :N :D 0 [REND "VRAI]
  RENDS DIV? :D + 2
FIN
```

La procédure PREM? est un prédicat (elle dit si le nombre est premier) qui utilise le prédicat DIV? (est-il divisible par ... ?) qui tourne éventuellement jusqu'à la racine carrée.

Comme précédemment, on retrouve facilement la liste des nombres premiers jusqu'à un nombre donné, en utilisant un compteur qu'on fait tourner à partir de 1, jusqu'à ce nombre final (la "moulinette LISTPREM1) et en rangeant les nombres premiers trouvés au fur et à mesure.

```
POUR LISTEPREM :F
  DONNE "LP []
  SOIT "K 1
  SI OU EGAL? :F 0 EGAL? :F 1 [REND :LP] [REND LISTEPREM1 :K]
FIN
```

```
POUR LISTEPREM1 :K
  SI PLG? :K :F [REND :LP]
  SI PREM? :K [DONNE "LP PH :LP :K]
  RENDS LISTEPREM1 :K + 1
FIN
```

### 3. Les nombres parfaits

En utilisant la procédure LISTE\_DIVISEURS, on peut savoir si un nombre est parfait, il suffit d'écrire une procédure qui va donner la somme des éléments d'une liste :

```
POUR SOMME_LISTE :LISTE
  SI VIDE? :LISTE [REND 0]
  RENDS SOMME PREM :LISTE SOMME_LISTE SP :LISTE
FIN
```

Je pense qu'elle se comprend sans explication supplémentaire, voir le lexique pour les primitives.

Et donc pour savoir si un nombre est parfait:

```

POUR PARFAIT? :NOMBRE
RENDS EGAL? SOMME_LISTE SD LISTE_DIVISEURS :NOMBRE
:NOMBRE
FIN

```

Cette procédure est loin d'être parfaite pour trouver les nombres parfaits, elle est très lente et je n'ai pu trouver par cette technique que les 4 premiers.<sup>1</sup>

## 4. Les congruences : Calcul de la clé RIB d'un compte chèque

On trouve sur le Net, la méthode pour vérifier la clé Rib d'un compte chèque. Le travail est intéressant parce que si on se tourne vers un bon vieux tableur, il reste en rade parce que le numéro de CC dépasse les limites de calcul et vous renvoie donc un nombre en écriture avec exposant. D'où l'intérêt d'utiliser les congruences. Je m'aperçois que je ne vous ai pas donné la méthode, oh elle est simple (?!), il suffit de multiplier votre numéro de CC par 100, de diviser ce nombre par 97, de vous occuper seulement du reste et de chercher son complément à 97. Il fallait y penser. Donc si on applique cela dans un tableur ça peut donner cela :

	A	B	C	D	E	F
1	Tapez votre numéro de compte en banque--->	167070006516519032343	16707000651651903234300			
2						
3		0	0	0		
4		1	0	0		
5		2	3	9		
6		3	30	4	120	
7		4	9	3	27	
8		5	90	2	180	
9		6	27	3	81	
10		7	76	0	0	
11		8	81	9	729	
12		9	34		34	
13		10	49		245	
14		11	5	6	30	
15		12	50	1	50	
16		13	15	5	75	
17		14	53	6	318	
18		15	45	0	0	
19		16	62	0	0	
20		17	38	0	0	
21		18	89	7	623	
22		19	17	0	0	
23		20	73	7	511	
24		21	51	6	306	
25		22	25	1	25	
26						
27						
28						
29						
30						
31						
32						
33						
34						

Mais bon, je ne suis pas là pour faire de la pub à Bilou, donc voyons comment avec Logo on peut résoudre cela:

Tout d'abord il faut écrire une procédure qui va nous donner la congruence modulo 97 d'une centaine, d'un millier etc... ça c'est facile :

<sup>1</sup> pour d'autres explications voir la notice historique en annexe.

```

POUR CONGRU :A
  SI EGAL? :A 1 [RENDS 1]
  RENDS RESTE PROD 10 CONGRU DIFF :A 1 97
FIN

```

Essayez CONGRU 3, ça donne 3, bien !!! CONGRU 4 ça donne 30 parfait. Vous avez les congruences dans le tableau Excel.

Bon, maintenant il suffit de mettre en face l'un de l'autre, une liste constituée de votre numéro de compte chèque (séparez les chiffres par un espace et n'oubliez pas les deux zéros à la fin) et de faire un produit terme à terme puis d'additionner. Pour que cela se fasse tout seul il suffit de créer la procédure SOM\_PROD.

```

POUR SOM_PROD :A :B
  SI VIDE? :A [RENDS 0]
  RENDS SOMME PROD PREM :A PREM :B SOM_PROD SPREM :A SPREM :B
FIN

```

Mais il nous faut aussi une procédure qui fabrique la liste des congruences à l'envers.

```

POUR LISTE_CONG :B :C
  SI PLG? :B :C [RENDS ""]
  RENDS PH LISTE_CONG :B + 1 :C CONGRU :B
FIN

```

Regardez encore comment est écrite cette procédure, elle renvoie la liste des congruences en partant de la plus haute vers la plus basse.

On a presque tout pour travailler. Il suffit de ranger cela dans des variables:

```

DONNE "B [ 1 6 7 0 7 0 0 0 6 5 1 6 5 1 9 0 3 2 3 4 3 0 0 ]
DONNE "A LISTE_CONG 1 COMPTE :B
EC 97 - RESTE SOMPROD :A :B 97

```

Et ça devrait vous afficher 32 comme sur la feuille d'Excel.

## 5. Calcul des éléments de la suite de Fibonacci

On peut reprendre la définition de la suite et écrire une procédure brutale:

```

POUR FIBO1 :A
  SI MEMBRE? :A [1 2] [RENDS 1]
  RENDS SOMME FIBO1 :A - 1 FIBO1 :A - 2
FIN

```

Tapez FIBO1 10 et laborieusement Logo vous affiche 55. Les récursivités sont longues à remonter et les calculs sont très lents, c'est pourquoi je vous conseille cette solution dont l'explication est donnée en annexe.

```

POUR FIB :I :A :B
  SI :I = 1 [RENDS :B]
  RENDS FIB :I - 1 :B :A + :B
FIN

```

```

POUR FIBO :I
  RENDS FIB :I 0 1
FIN

```

Vous pouvez vous permettre d'afficher FIBO 30, c'est pas mal. Et si vous voulez la liste des nombres de Fibo, il suffit d'écrire cette récursivité non terminale:

```

POUR LISTE_FIBO :N
  SI EGAL? :N 0 [STOP]
    LISTE_FIBO :N - 1
  EC FIBO :N
FIN

```

Maintenant que nous avons la procédure FIBO efficace, il nous est possible de vérifier la convergence de  $u_{n+1} / u_n$ .

```

POUR CONV :D :F
  SI EGAL? :D :F [STOP]
    EC DIV FIBO :D + 1 FIBO :D
  CONV :D + 1 :F
FIN

```

Tapez CONV 1 25 et vous voyez que ça converge vers  $\frac{1}{2}(\sqrt{5} + 1)$ .

De même vous pouvez vérifier que  $u_{n+3} \times u_n - u_{n+2} \times u_{n+1}$ . Essayez pour différents nombres. Il serait facile de transformer FIBO en LUCAS.

Et enfin pour terminer ce chapitre sur l'arithmétique et Logo, une propriété que j'ai trouvée dans "Turtle Geometry", puis au paragraphe suivant une idée pour trouver ces fameux nombres de Bezout (ou résoudre des équations diophantiennes ?).

FIBO PGCD :A :B = PGCD FIBO :A FIBO :B. Vous pouvez la vérifier sur quelques nombres. La démonstration est parait-il "hard".

## 6. Égalité de Bezout et récursivité non terminale.

Reprenons l'algorithme d'Euclide vu au début de ce chapitre d'arithmétique en le suivant sur un exemple avec les nombres 64 et 36.

“Descendons” l'algorithme à partir de 64 et 36 pour trouver leur PGCD 4 (par soustractions successives), puis essayons de “le remonter” pour trouver deux nombres p et q tels que :

$$p \cdot 64 + q \cdot 36 = 4.$$

Nous aurons ainsi trouvé ‘une égalité’ de Bezout.

Vous verrez qu'il n'y a pas unicité du couple trouvé (p,q) à partir des deux nombres 64 et 36 (nommés R et N ci dessous).

<b>R</b>	<b>N</b>	<b>p</b>	<b>q</b>	<b>pR + qN = d</b>
<b>64</b>	<b>36</b>	<b>-5</b>	<b>9</b>	<b>-5 * 64 + 9 * 36 = 4</b>
64-36	36			
<b>28</b>	<b>36</b>	<b>-5</b>	<b>4</b>	<b>-5 * 28 + 4 * 36 = 4</b>
28	36-28			
<b>28</b>	<b>8</b>	<b>-1</b>	<b>4</b>	<b>-1 * 28 + 4 * 8 = 4</b>
28-8	8			
<b>20</b>	<b>8</b>	<b>-1</b>	<b>3</b>	<b>-1 * 20 + 3 * 8 = 4</b>
20-8	8			
<b>12</b>	<b>8</b>	<b>-1</b>	<b>2</b>	<b>-1 * 12 + 2 * 8 = 4</b>
12-8	8			
<b>4</b>	<b>8</b>	<b>-1</b>	<b>1</b>	<b>-1 * 4 + 1 * 8 = 4</b>
4	8-4			
<b>4</b>	<b>4</b>	<b>0</b>	<b>1</b>	<b>0 * 4 + 1 * 4 = 4</b>

Quand vous avez fini de “descendre l'algorithme”, vous obtenez deux valeurs égales pour R et N. Elles sont le PGCD de tous les couples (R;N) rencontrés.

Pour “faire la première égalité de Bezout”, à la dernière ligne de la descente, vous pourrez affecter par exemple p et q à 0 et 1.

$$0 \cdot 4 + 1 \cdot 4 = 4$$

Vous pourriez faire d'autres choix que le couple (0;1) pour (p;q).

Puis vous “remontez l'algorithme” ligne à ligne, en vous souvenant des changements effectués dans la descente. Par exemple, si en descendant du niveau n au niveau n+1, vous avez remplacé R par R-N en gardant N, en remontant du niveau n+1 au niveau n, vous garderez p et changerez q en q-p.

R	N	p	q-p	pR + (q-p)N = d	<- niveau n
R-N	N	p	q	p(R-N) + qN = d	<- niveau n+1

Par contre, pour le changement de N en N-R en descente avec R conservé, en remontant, vous garderez q et vous changerez p en p-q .

$$\begin{array}{rcccccc} R & N & p-q & q & (p-q)R + qN & = d & \leftarrow \text{niveau } n \\ R & N-R & p & q & pR + q(N-R) & = d & \leftarrow \text{niveau } n+1 \end{array}$$

Pour faire tout ça, il vous suffira de modifier EUCLIDE de la sorte :

```
POUR EUCLIDE :N :R
  SI EGAL? :R :N [DONNE "P 0 DONNE "Q 1 DONNE "PGCD :N STOP]
  SI PLG? :N :R [ EUCLIDE :N - :R :R DONNE "Q :Q - :P] [EUCLIDE :N :R - :N DONNE "P
:P - :Q]
  EC PH PH PH PH PH PH PH PH :P "*" :N "+" :Q "*" :R "=" :PGCD
FIN
```

Avec la magie de la récursivité non terminale, c'est le dernier "clone" EUCLIDE (vous pouvez le nommer "EUCLIDE 4 4") qui se charge de dire que P vaut 0, que Q vaut 1, et que le PGCD est le nombre N, de valeur 4, que lui a passé son collègue précédent (nommé "EUCLIDE 4 8").

EUCLIDE 4 4 écrit les trois nombres dans trois variables globales.

Quand EUCLIDE 4 8 reçoit le message qui remonte, il regarde ce qu'il lui restait à faire lorsqu'il avait appelé EUCLIDE 4 4, juste après qu'il ait fait son test.

Il s'était dit :

"4 est il plus grand que 8 ? non donc j'appelle EUCLIDE 4 4"

Maintenant que EUCLIDE 4 4 lui a rendu le résultat de son travail, il reste à faire à EUCLIDE 4 8 la fin du test (le "sinon"). Il met dans "P le résultat de :P - :Q soit 0 - 1 (ce qui fait -1). Il ne modifie pas la variable "Q, qui vaut donc toujours 1, ni "PGCD qui est 4. Puis il écrit, en utilisant ses variables locales "N et "R qui valent 4 et 8 :

$$-1*4+1*8 = 4$$

Cette fois EUCLIDE 4 8 a bien fini ... au tour de EUCLIDE 12 8 de faire son travail ... Regardez, ils ont tout fait !

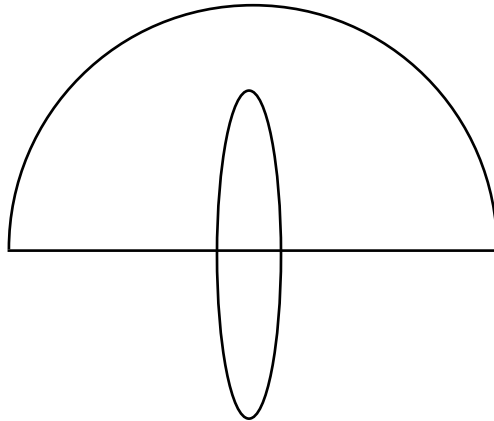
```
? EUCLIDE 64 36
-1 * 4 + 1 * 8 = 4
-1 * 12 + 2 * 8 = 4
-1 * 20 + 3 * 8 = 4
-1 * 28 + 4 * 8 = 4
-5 * 28 + 4 * 36 = 4
-5 * 64 + 9 * 36 = 4
```

Fabuleux ? non ?

Vous pourrez améliorer votre équipe de travailleurs en leur apprenant à utiliser les divisions successives à la place des soustractions successives. Vous pouvez également souhaiter qu'ils vous rendent la liste [:P :Q] sans écrire ... Proposer un démarrage du premier couple (p,q) avec un choix aléatoire de l'un des deux ...

# Barycentre et théorème de Guldin

Petit rappel: pour une surface de révolution, son aire est égale à la longueur de la ligne qui engendre la surface par la longueur du cercle engendré par le centre de gravité de la ligne. C'est le théorème de Guldin<sup>2</sup>. Ce qui donne pour la sphère ci-dessous:  
Aire = demi-périmètre du cercle x périmètre du cercle décrit par le centre de gravité du demi-cercle.



Voyons comment la tortue peut vérifier ce théorème.  
Nous allons tout d'abord faire décrire à la tortue un demi-cercle. Rien de plus facile:

```
POUR CERCL3
DONNE "C [[0 0]]
TD 1
REPETE 90 [AV 5 TD 2 DONNE "C PH :C LISTE POS "]
FIN
```

Lancez la procédure CERCL, Logo s'occupe de dessiner et va placer dans une liste de listes (variable C) les coordonnées des différents points sur lesquels elle passe. Si vous demandez l'écriture de C, ça devrait vous donner à peu près cela:

```
[0 0] [0,08726203218640194 4,999238475781965] [0,34894181340109753
9,992386149554818] [0,7847205271393705 14,973359640013541].....
```

Une liste de 91 listes.

Il nous faut maintenant écrire les procédures pour trouver le centre de gravité de cette ligne, c'est à peu près l'isobarycentre de ces 91 points.

```
POUR ISOBARYCENTRE :A
RENDS PH DIV SOM X :A COMPTE :A DIV SOM Y :A COMPTE :A
FIN
```

Pour obtenir l'abscisse du centre de gravité, il suffit de sommer les abscisses des différents points rencontrés par la tortue, il nous faut donc écrire deux procédures, l'une qui récupère la liste des abscisses, l'autre qui somme les nombres d'une liste.



<sup>2</sup> Guldin Paul né en 1577 - mort le 3 nov 1643.

<sup>3</sup> On ne peut pas appeler une procédure CERCLE parce que ce mot est une primitive du langage LOGO, elle dessine un cercle de rayon donné et de centre donné.

La liste des abscisses sera écrite de la façon suivante:

```
POUR X :A
  SI VIDE? :A [RENDS "]
  RENDS PH PREM PREM :A X SP :A
FIN
```

Idem pour la liste des y mais on prend dans chaque élément de la liste le dernier.

```
POUR Y :A
  SI VIDE? :A [RENDS "]
  RENDS PH DER PREM :A Y SP :A
FIN
```

Il suffit décrire une procédure qui additionne les nombres d'une liste:

```
POUR SOM :A
  SI VIDE? :A [RENDS 0]
  RENDS SOMME PREM :A SOM SP :A
FIN
```

Comme nous voulons seulement vérifier la validité du théorème de Guldin, nous allons procéder de la façon suivante. Comme on sait déjà que l'aire est égale à  $4\pi R^2$ . Le quotient du diamètre du cercle par le rayon du cercle engendré par le centre de gravité doit être égal à  $\pi$ . Il nous suffit donc de demander à Logo de faire ce calcul en tapant:

```
VE
CERCL
EC DIV POS DER ISOBARYCENTRE :C
Allez-y et vous verrez c'est pas si mal.
```

Naturellement, cette pratique sera plus intéressante à utiliser lorsque la figure engendrée sera plus complexe.

Il est aussi possible d'étudier la procédure BARYCENTRE dans un cadre plus géométrique, dans ce cas on écrirait la procédure barycentre de la façon suivante:

```
POUR BARYCENTRE :A :B
  SI EGAL? COMPTE :A 2 [RENDS BARY :A :B]
  DONNE "BARY_PROVISOIRE
  BARYCENTRE LISTE PREM :A PREM SP :A LISTE PREM :B PREM SP :B
  RENDS BARYCENTRE MP :BARY SP SP :A MP SOMME PREM :B PREM SP :B SP SP :B
FIN
```

MP *item1 item2* est une primitive Logo qui place en première place *item1* d'une liste *item2*. Donc si je lis bien du Logo, ça signifie qu'on recommence la procédure BARYCENTRE avec une nouvelle liste dans laquelle les coordonnées du barycentre des deux premiers points remplacent celles des deux premiers points (les deux premiers points sont supprimés SP SP :A) et dans la liste des coefficients, les deux premiers coefficients sont remplacés par leur somme.

Il suffit d'écrire la procédure BARY :A :B qui va rendre les coordonnées du barycentre de deux points affectés des coefficients placés dans la liste B.

L'intérêt de travailler de cette façon-là c'est qu'on généralise la procédure BARYCENTRE mais aussi que l'on peut visualiser le chemin engendré par les différents centres de gravité transitoire. Voir à cet effet la brochure IREM: LOGO et géométrie — Alain Bois et Jacques Delgoulet 1986.

## Annexe Histoire des nombres parfaits

renseignements tirés de [http://www-groups.dcs.st-and.ac.uk/~history/HistTopics/Perfect\\_numbers.html](http://www-groups.dcs.st-and.ac.uk/~history/HistTopics/Perfect_numbers.html)

### ☛ Euclide

Les 4 premiers nombres parfaits sont connus d'Euclide (voir la proposition 36 du livre IX des *Éléments*). Ils sont décrits comme étant le produit de la somme de puissances de 2 (à condition que cette somme soit un nombre premier) par la dernière puissance utilisée:

$$\begin{array}{ll} 1 + 2 = 3 & 3 \times 2 \text{ est parfait} \\ 1 + 2 + 4 = 7 & 7 \times 4 \text{ est parfait} \\ 1 + 2 + 4 + 8 = 15 & 15 \times 8 \text{ est imparfait} \end{array}$$

Par contre :

$$1 + 2 + 4 + 8 + 16 = 31 \text{ et } 31 \times 16 \text{ est parfait ...}$$

Avec nos notations modernes nous écrivons un résultat important:(appelé algorithme d'Euclide)

$$(2^n - 1) \times 2^{n-1} \text{ est parfait si } 2^n - 1 \text{ est premier.}$$

### ☛ Nicomaque de Gérase

Nicomaque de Gérase (100 ap. JC) dans *Introduction Arithmetica* donne une classification des nombres comme suit:

- les nombres surabondants
- les nombres déficients
- les nombres parfaits

Mais pour lui ces nombres sont régis par des considérations morales:

*... dans le cas où il y surabondance, cela produit de l'excès, du trop plein, de l'exagération et de l'abus. Dans le cas contraire, il y a défaut, privation, demande et insuffisance. Enfin dans le meilleur des cas, il y a vertu, juste mesure, propriété, beauté et toute chose de cette sorte.*

S'y ajoutent des considérations sur la monstruosité:

*10 bouches, 10 lèvres, 3 rangées de 10 dents sont une illustration des nombres surabondants. Un œil, un bras avec une main ayant moins de 5 doigts illustrent les nombres déficients.*

Sans démonstration, il donne 5 propositions:

1.  $\square$  nième nombre parfait a n chiffres
2.  $\square$ es nombres parfaits sont pairs
3.  $\square$ ous les nombres parfaits se terminent par 6 ou 8 et cela alternativement.
4.  $\square$ 'algorithme d'Euclide permet de les trouver
5.  $\square$  y a une infinité de nombres parfaits

La première et la 3ème proposition sont fausses, le 5ème nombre parfait est 33550336 et le 6ème est 8589869056. Il fournit une méthode nouvelle pour les trouver:

Vous écrivez sur une ligne les puissances de 2

1	2	4	8	16	32	64	128	256	512	1024
---	---	---	---	----	----	----	-----	-----	-----	------

Vous ajoutez les nombres au fur-et-à-mesure, si vous trouvez un nombre premier, vous le multipliez par la dernière puissance de 2 prises pour l'addition.

Saint-Augustin (354-430) note dans la "cité de Dieu" que 6 est parfait par lui-même, Dieu n'a pas créé le monde en 6 jours donc 6 est parfait mais Dieu a créé le monde en 6 jours parce que 6 est parfait.

☛ ibn Ibrahim ibn Fallus (1194-1239) écrit dans un traité basé sur *Introduction à l'arithmétique* de Nicomaque qu'il a trouvé les dix premiers nombres parfaits. Il s'avéra plus tard que les 7 premiers seulement étaient corrects.

- ☛ Charles de Bovelles croit en l'assertion suivante  $(2^n - 1) \times 2^{n-1}$  est parfait si n est impair.
- ☛ Le 5ème nombre parfait fut trouvé dans un manuscrit de 1461, il se trouve aussi dans un manuscrit de Regiomontanus. Le 5ème et le 6ème sont inscrits ensemble dans un autre manuscrit écrit dans les environs de 1460.
- ☛ En 1536, Hudalrichus Regius montre que  $2^{11} - 1$  n'est pas premier puisque décomposable en  $23 \times 89$  mais par contre  $2^{13} - 1$  est premier.
- ☛ En 1603, Cataldi prouve que  $2^{17} - 1$  est premier et donc que 589869056 est parfait mais aussi que  $2^{19} - 1$  l'est aussi et donc que 137438691328 est parfait. Il prétend que pour  $p = 2, 3, 5, 7, 13, 17, 19, 23, 29, 31, 37$   $2^p - 1$  est premier. Malheureusement sur les 4 derniers nombres un seul est correct.
- ☛ Descartes dans une lettre de 1638 à Mersenne lui dit son doute de trouver un nombre parfait impair.
- ☛ Fermat prouve que  $2^{23} - 1$  et  $2^{37} - 1$  sont factorisables dans une lettre envoyée en juin 1640 à Mersenne.
- ☛ Mersenne annonce que les nombres de la forme  $2^n - 1$  sont premiers pour  $n = 2, 3, 5, 7, 13, 17, 19, 31, 67, 127, 257$  et c'est tout.
- ☛ Euler en 1732 prouve que  $(2^{31} - 1) \times 2^{30} = 2305843008139952128$  est parfait.
- ☛ Lucas en 1876 montre que Mersenne s'est trompé  $2^{67} - 1$  n'est pas premier mais ne trouve pas sa factorisation par contre  $2^{127} - 1$  l'est comme l'avait prédit Mersenne.
- ☛ En 1903, Cole lors d'une conférence de l'AMS écrit au tableau:  
 $2^{67} - 1 = 147573952589676412927$   
 puis commence la multiplication à la main de 761838257287 par 193707721 et trouve 147573952589676412927. Ce qui lui valut une "standing ovation".  
 Plusieurs erreurs de Mersenne furent trouvées. En 1911 Powers montre que  $2^{88} (2^{89} - 1)$  est parfait (ce fut le dernier nombre parfait trouvé sans l'aide de l'ordinateur), puis quelques années plus tard que  $2^{101} - 1$  est premier. En 1922 Kraitchik montre que  $2^{257} - 1$  n'est pas premier. A l'heure actuelle le 39ème nombre parfait connu (décembre 2001) est  $2^{13466916} \times (2^{13466917} - 1)$  et il a plus de 4 millions de chiffres en notation décimale. Mais l'histoire des nombres parfaits n'est pas finie ...

Les auteurs de l'article sont *J J O'Connor* et *E F Robertson*