Maple

Lycée Clemenceau - MP⋆ 2013-2014

Version du 24 novembre 2013

Une dosette de MAPLE en MP⋆

Guillaume CONNAN

UNIVERSITÉ DE NANTES

IUT NANTES Informatique

SOMMAIRE

1 X : info B **3**

1.1 L'algo à l'X . . . 4
1.2 Le tutoriel de M. Stainer . . . 4
1.3 X/Cachan 2013 . . . 31
1.4 X/Cachan 2010 . . . 35

2 L'aventure géométrique **43**

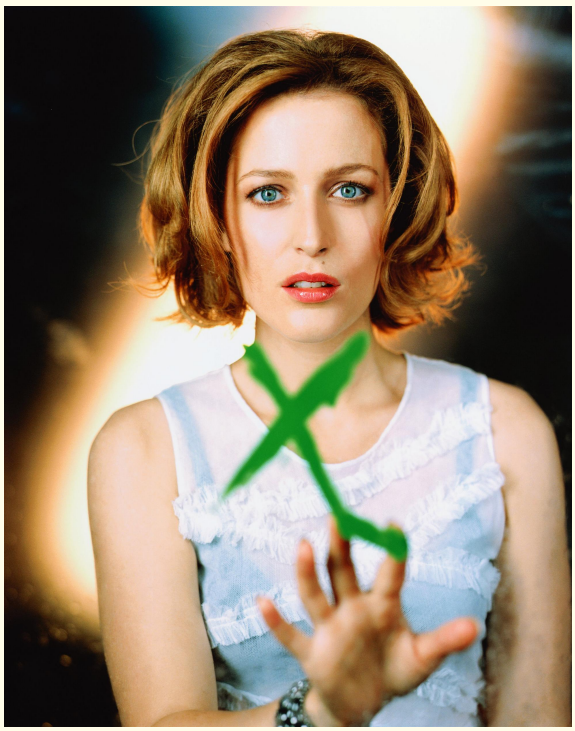
2.1 Et le taupin créa la tortue . . . 44
2.2 Polygone . . . 45
2.2.1 Des expériences . . . 45
2.2.2 Un théorème . . . 46
2.3 Une tortue prédatrice . . . 46
2.4 Une tortue topologique . . . 48
2.5 Le tour du monde de la tortue . . . 49

3 Courbes elliptiques **50**

3.1 Un peu de lecture... . . . 51
3.2 Quelques résultats préliminaires . . . 51
3.3 Loi de groupe sur une cubique . . . 52
3.4 Factorisation d'entiers : méthode de LENSTRA . . . 54
3.5 Cryptographie . . . 56
3.6 Jokers . . . 57

TP N° 1

X : info B



Armé(e) d'un maple 12 avant-avant-dernière génération, vous allez vous attaquer au sujet X/Cachan des MP-SI débarqué à Palaiseau en juin dernier et vous pourrez alors attaquer seul(e) celui de 2010...mais vous n'êtes pas seul(e)...

1 L'algo à l'X

L'épreuve B d'informatique dure deux heures et n'est corrigée que pour les candidats admissibles.
Le choix du langage de programmation est libre.
Environ 60% des candidats choisissent Maple, 25% OCaml, 6% C ou C++, 4% Python et 3% des langages divers.
Nous illustrerons le sujet de 2013 en Maple car cela nous servira aussi à préparer l'épreuve de Math IIde Centrale.
Voici quelques remarques que l'on retrouve dans les rapports d'année en année.

- Il faut faire attention aux directives du sujet, par exemple s'il est indiqué que l'indexation des tableaux commence à zéro. Vous pouvez faire un autre choix en l'indiquant clairement, mais cela embêtera le correcteur....
- Il faut proscrire toutes les commandes de calcul formel.
- Le code mis en œuvre doit être simple et clair dans sa forme autant que dans sa présentation : indentation, passage à la ligne...
- N'hésitez pas à commenter votre code (avec # en Maple).
- Un ancien correcteur n'aimait pas la récursivité. Il faudra attendre le rapport 2013 pour voir ce qu'il en est du nouveau. Nous en reparlerons.
- Donnez des noms explicites aux fonctions intermédiaires que vous créez.
- Attention à utiliser des fonctions qui retournent les valeurs demandées et non pas leur affichage avec un petit message du style « *La réponse est : 5* ».
- N'oubliez pas d'initialiser vos variables.
- N'oubliez pas que les fonctions des questions précédentes servent souvent à créer celles des suivantes.
- Pensez à économiser la mémoire en évitant les évaluations répétées d'une même expression.
- Une erreur bizarre et fréquente : si votre indice de boucle est `i`, ne vous attendez pas à ce qu'une variable `ti` s'incrémente automatiquement avec `i`...
- Autre erreur fréquente et un peu bête : entre les indices `a` et `b`, il y a `b-a+1` valeurs !...

Le sujet de 2013 étudie la recherche de points fixes de fonctions à valeurs dans un ensemble fini.
Vous noterez que quatre questions parmi les quatorze ne demande pas de code mais des calculs de complexité ou des démonstrations mathématiques...mais un programme est lui-même une démonstration...
Le sujet est un peu plus difficile que d'habitude mais on y retrouve toujours la manipulation de tableaux. Il y a comme d'habitude des questions pouvant être considérées hors programme mais le hors programme semble être au programme à l'X et aux ENS...

2 Le tutoriel de M. Stainer

Voici à présent un rappel des principales fonctions Maple proposées par M. Stainer qui seront surtout utiles pour l'oral de Centrale et qu'il faudra donc conserver par devers vous.

Maple 12 - Interface et programmation

Quelques outils

Remarque sur les interfaces

L'interface Java, seule disponible dans la version 64 bits de Maple, propose une mise en forme automatique plutôt esthétique des expressions mathématiques, mais parfois difficile à débugger, par exemple lorsque des espaces indésirables se sont glissés... En cas de doute, on peut convertir la commande au format "**`1-D Math Input`**" via le menu "Format/Convert to". On peut aussi saisir la commande directement dans ce mode en cliquant sur le bouton "Text", à gauche sous la barre des onglets (**Ctrl+M** au clavier).
Dans la version 32 bits de Maple, on peut choisir l'interface "Classic Worksheet", qui est globalement au format "**`1-D Math Input`**".

Remarques sur la ponctuation

Les instructions sont séparées par **;** ou **:**. L'utilisation de **:** empêche l'affichage du résultat.
Plusieurs instructions peuvent figurer sur une seule ligne de commande, plusieurs lignes de commande peuvent être regroupées dans un *groupe d'exécution*. La touche **Entrée** lance l'exécution de toutes les instructions du groupe d'exécution (d'où son nom !) où se trouve le curseur (inutile notamment de placer le curseur en fin de ligne avant de taper **Entrée** !).

Raccourcis clavier utiles :

- **Crtl+T** fait passer en mode texte, **Ctrl+R** en mode "2-D Math Input" et **Ctrl+M** en mode "**`1-D Math Input`**"
- **Maj+Entrée** permet d'insérer un saut de ligne dans une commande sans lancer l'évaluation ; cela permet par exemple de mettre en page un programme
- **F4** permet de joindre le groupe d'exécution où se trouve le curseur et le suivant ; il faut savoir que taper **Entrée** lance l'exécution de toutes les commandes du groupe d'exécution (d'où son nom !) où se trouve le curseur (inutile notamment de placer le curseur en fin de ligne avant de taper **Entrée** !) ; les groupes d'exécution sont délimités par les crochets de la marge gauche ; **F3** scinde la commande courante en deux groupes d'exécution, à l'endroit où se trouve le curseur (cf. menu "Edit/Split or Join")
- **Ctrl+J** insère un nouveau groupe d'exécution au-dessous du groupe où se trouve le curseur, **Ctrl+K** en insère un au-dessus

Sur une ligne contenant un #, les caractères suivant ce # sont ignorés et peuvent donc servir de commentaires.

Le symbole % (*dito*) permet de rappeler les trois derniers résultats : % pour le dernier, %% pour l'avant-dernier et %%% pour l'antépénultième ! Attention, *dernier* se réfère ici à l'**ordre chronologique** des exécutions et non à l'ordre de haut en bas sur la feuille...

Encadrer une (sous-)expression par des apostrophes (*quotes*) a pour effet de différer son évaluation d'un niveau :

```
> 'sin'(Pi);%;
```

$$\sin(\pi) \qquad \textbf{(1.2.1)}$$

$$0$$

Les guillemets (") permettent de former une chaîne de caractères :

```
> "Voici une chaîne";
```

"Voici une chaîne" **(1.2.2)**

La virgule sépare les différents éléments d'une *séquence*. Les séquences sont omniprésentes : les séquences entre parenthèses servent d'argument aux appels de fonctions ; les *listes* sont les séquences entre crochets, les *ensembles* sont les séquences entre accolades (dans un ensemble, les éléments n'apparaissent qu'une fois et on ne contrôle plus l'ordre des éléments).
La séquence vide est représentée par le mot réservé **`NULL`**.
L'affectation est réalisée par **:=**. Dans Maple 12, on peut affecter des valeurs à plusieurs variables avec un seul **:=**, en utilisant des séquences :

```
> x,y:=1,2;
```

$$x, y := 1, 2 \qquad \textbf{(1.2.3)}$$

Type et opérandes d'un objet

Les objets manipulés par Maple ont un *type* renvoyé par la fonction **whattype** :

```
> whattype('sin'(Pi));whattype(sin(Pi));
```

function

integer **(1.3.1)**

type(expr,t) renvoie *true* ou *false* selon que **expr** est ou non de type **t** (voir dans la feuille d'aide de **type** les types prédéfinis).

Les expressions sont stockées sous forme d'un arbre dont la racine correspond au type de l'objet. **nops(expr)** renvoie le nombre d'opérandes (au premier niveau), **op(expr)** renvoie la séquence des opérandes. Pour **i** compris entre **1** et **nops(expr)**, **op(i,expr)** renvoie le **i**-ème opérande et **op(-i,expr)** le **i**-ème opérande à partir du dernier.

En général, **op(0,expr)** renvoie le type de l'expression ; lorsque le type est *function*, **op(0,expr)** renvoie le nom de la fonction. Il faut noter que le type *function* pour Maple représente un **appel de fonction** et non une fonction au sens mathématique du terme (une telle fonction est du type *symbol* !).

```
> p:=a*(b+1)/c:op(0,p);op(p);
```

`*`

$$a,\ b+1,\ \frac{1}{c} \qquad \textbf{(1.3.2)}$$

```
> f:=u(x,y);whattype(f);op(0,f);op(f);
```

$$f := u(x, y)$$

function

$$u$$

$$x,\ y \qquad \textbf{(1.3.3)}$$

```
> vg:=(x,y)->x+y-x*y;whattype(g);op(g);
```

$$g := (x, y) \rightarrow x + y - x\,y$$

symbol

$$(x, y) \rightarrow x + y - x\,y \qquad \textbf{(1.3.4)}$$

subsop(i=newexpr,expr) remplace le **i**-ème opérande de **expr** par **newexpr** :

```
> subsop(1=a+1,f);
```

$$u(a+1,\ y) \qquad \textbf{(1.3.5)}$$

À ne pas confondre avec **subs(old=new,expr)**, qui remplace dans **expr** toutes les occurrences de la sous-expression **old** par **new** :

```
> p:=(a+b+1)/2*(a+b);subs(a+b=s,p);
```

$$p := \frac{1}{2}\ (a+b+1)\ (a+b)$$

$$\frac{1}{2}\ (a+b+1)\ s \qquad \textbf{(1.3.6)}$$

Le résultat précédent peut paraître surprenant : en fait **subs** ne remplace que des sous-expressions complètes ; on peut parfois utiliser un subterfuge :

```
> subs(a=s-b,p);
```

$$\frac{1}{2}\ (s+1)\ s \qquad \textbf{(1.3.7)}$$

Les opérandes d'une liste ou d'un ensemble en sont les éléments ; mais, pour appliquer à une séquence les fonctions **op**, **nops** ou **subsop**, il est nécessaire de la convertir en liste :

```
> s:=a,b,c;nops(s);nops([s]);
```

$$s := a,\ b,\ c$$

```
Error, invalid input: nops expects 1 argument, but received 3
```

3 **(1.3.8)**

Quelques commandes itératives pré-programmées

Attention ! Les outils présentés dans cette section ne doivent pas être utilisés pour l'épreuve d'algorithmique de l'X…

seq, add, mul et $, sum, product

seq et **$** permettent de construire des séquences, **add** et **sum** effectuent des sommes, **mul** et **product** des produits ! **seq**, **add**, **mul** sont plus efficaces pour les calculs numériques, **$**, **sum** et **product** permettent les calculs formels.

```
> add(k,k=1..n);sum(k,k=1..n);
Error, unable to execute add
```

$$\frac{1}{2}\,(n+1)^2-\frac{1}{2}\,n-\frac{1}{2} \qquad \textbf{(1.4.1.1)}$$

La syntaxe générale pour l'opérateur **$** est **expr$i=a..b** ; le résultat est alors la séquence obtenue en substituant successivement **a**, **a+1**,..., **b** à **i** dans **expr**. Il existe deux raccourcis utiles en pratique : **$a..b** pour **i$i=a..b** et **expr$n** pour **expr$i=1..n**, lorsque **expr** ne dépend pas de **i** :

```
> {$3..12};diff(tan(x),x$3);
```

$$\{3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$$

$$2\,(1+\tan(x)^2)^2+4\,\tan(x)^2\,(1+\tan(x)^2) \qquad \textbf{(1.4.1.2)}$$

seq, **add** et **mul** acceptent deux syntaxes : si l'appel est effectué avec **(expr,i=a..b)**, alors **i** reçoit successivement les valeurs **a**, **a+1**,..., **b** ; si l'appel est effectué avec **(expr,i=X)**, où **X** est une expression, alors **i** est remplacé successivement par les différents opérandes de **X** :

```
> seq(i^2,i=1..5);add(u^3,u=a+2-3*b);
```

$$1, 4, 9, 16, 25$$

$$8+a^3-27\,b^3 \qquad \textbf{(1.4.1.3)}$$

sum et **product** acceptent la première forme d'argument ci-dessus, mais pas la seconde ; pour plus de détails, voir les feuilles d'aide.

map, select et remove

En général, **map(f,expr)** remplace chaque opérande x_i de **expr** par $f(x_i)$. Plus généralement, **map(f, expr,y)** remplace chaque opérande x_i de **expr** par $f(x_i, y)$ et ainsi de suite avec davantage d'arguments le cas échéant :

```
> restart:map(x->x^2,a+b+c);map(f,[a,b,c],y,z);
```

$$a^2+b^2+c^2$$

$$[f(a, y, z), f(b, y, z), f(c, y, z)] \qquad \textbf{(1.4.2.1)}$$

Une exception dans le cas où **expr** est un tableau (donc notamment dans le cas particulier des vecteurs et des matrices) : f est alors appliquée à chaque élément du tableau.

```
> map(k->k+x,Matrix(3,3,(i,j)->i+j-1));
```

$$\begin{bmatrix} 1+x & 2+x & 3+x \\ 2+x & 3+x & 4+x \\ 3+x & 4+x & 5+x \end{bmatrix} \qquad \textbf{(1.4.2.2)}$$

select (resp. **remove**) utilisent la même syntaxe, mais f doit renvoyer pour chaque opérande *true* ou *false* et le résultat est une expression du même type, dont les opérandes sont ceux pour lesquels f a renvoyé *true* (resp. *false*) :

```
> select(isprime,[$1..30]);remove(type,a+1-b-4,integer);
```

$$[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]$$

$$a-b \qquad \textbf{(1.4.2.3)}$$

Les trois valeurs booléennes de Maple - is et assume

Pour utiliser sereinement les commandes précédentes et les instructions conditionnelles définies ci-dessous, il faut savoir que Maple utilise, outre les deux valeurs booléennes *true* et *false* bien connues, une troisième valeur, représentée par le mot réservé *FAIL*, que l'on pourrait traduire par "je ne sais pas", ou encore "l'algorithme utilisé ne permet pas de conclure".
L'utilisation des tests avec des variables libres peut conduire à des erreurs ou même à des réponses discutables :

```
> x:='x':L:=[x,0,1]:select(k->k<>0,L);select(k->k>0,L);
```

$$[x, 1]$$

```
Error, selecting function must return true or false
```

```
> if x<0 then -1 else 1 fi;
Error, cannot determine if this expression is true or false: x < 0
```

Maple est sûr que *x* est non nul, mais ne peut déterminer son signe ...
Il est possible, pour éviter cette dernière erreur, d'utiliser la fonction **is**, qui renvoie *FAIL* lorsque Maple ne peut conclure :

```
> if is(x<0) then -1 else 1 fi;
  if is(x>=0) then 1 else -1 fi;
```

$$1$$

$$-1 \qquad \textbf{(1.5.1)}$$

FAIL est considéré comme une réponse négative : il faut donc être attentif dans le choix des tests et l'utilisation de **else** !
L'instruction suivante, incongrue dans le cadre d'une logique binaire, est à méditer :

```
> if is(x<0) then -1 elif is(x>=0) then 1 else 'sgn'(x) fi;
```

$$sgn(x) \qquad \textbf{(1.5.2)}$$

La fonction **is** est souvent utilisée en association avec la fonction **assume**, qui permet de soumettre une variable à des hypothèses :

```
> is(x>0);assume(x>2);is(x>0);
```

$$false$$

$$true \qquad \textbf{(1.5.3)}$$

Voir la feuille d'aide de **assume** pour plus de détails.

Principales instructions de programmation

On remarquera que, lors de la validation, Maple 12 remplace les "raccourcis" **fi**, **od** et **end** par **end if**, **end do** et **end proc**.

Instructions conditionnelles

La structure de base est :

if *cond* **then** $instructions_v$ **else** $instructions_f$ **fi**

Pour exécuter cette instruction, Maple effectue tout d'abord l'évaluation de *cond*, qui doit fournir un résultat booléen : si ce résultat est *true*, alors les $instructions_v$ sont exécutées ; si ce résultat est *false* ou *FAIL* alors les $instructions_f$ sont exécutées. La clause **else** est facultative.

Lorsqu'il y a plusieurs cas à envisager, on peut imbriquer des instructions conditionnelles ou bien ajouter une (ou des) clause(s) **elif** à la structure précédente selon le schéma suivant :

if $cond_1$ **then** $instructions_1$
elif $cond_2$ **then** $instructions_2$
elif ...
else $instructions_f$
fi

elif est interprété comme **else if**, mais son utilisation permet d'éviter les ambiguités qui peuvent apparaître en présence de plusieurs **else** : ici, la clause **else**, facultative, introduit les instructions qui sont exécutées si les réponses à tous les tests sont négatives. De plus, l'utilisation de cette structure permet de conclure l'instruction par un seul **fi**.

Instructions itératives

La boucle for - from

Sa syntaxe est la suivante :

for *nom* **from** *début* **by** *pas* **to** *fin* **while** *cond* **do** *instructions* **od**

Pour exécuter cette instruction, Maple commence par évaluer les expressions *début*, *fin* et *pas*, puis affecte à la variable désignée par *nom* les valeurs successives *début* , *début* + *pas* , *début* + 2 *pas* ,... ; tant que la valeur de *nom* reste inférieure ou égale à *fin* et que l'évaluation de *cond* donne *true*, les *instructions* sont exécutées.

Chacune des clauses **for**, **from**, **by**, **to**, **while** peut être omise. En l'absence de **for**, une "variable muette" est utilisée à la place de *nom* (ce qui suppose que le reste de l'instruction ne dépend pas de *nom*) ; par ailleurs, les valeurs par défaut de *début*, *pas*, *fin* et *cond* sont respectivement 1,1, ∞ et *true*.

do et **od** sont obligatoires mais la suite des *instructions* peut être vide.

À la sortie de la boucle, *nom* contient la dernière valeur testée.

Voici quelques exemples :

```
> for i to 3 do i^2 od;
```

1

4

9 **(2.2.1.1)**

Pour trouver le plus petit nombre premier supérieur à 1959 :

```
> for i from 1959 by 2 while not isprime(i) do od:i;
```

1973 **(2.2.1.2)**

Attention ! Ce mélange de **for** et **while** est déconseillé dans les épreuves classiques d'algorithmique...

La boucle while

C'est en fait un cas particulier du précédent, sous la forme :

while *cond* **do** *instructions* **od**

Les *instructions* sont exécutées tant que l'évaluation de *cond* donne *true*.

Cherchons par exemple à 0.0001 près par défaut la moyenne arithmético-géométrique de 1 et 2 :

```
> a:=1.0:b:=2.0:v
  while (b-a)>.0001 do
  c:=sqrt(a*b);b:=.5*(a+b);a:=c;
  od:
  a;
```

1.456791014 **(2.2.2.1)**

La boucle for - in

Sa syntaxe est la suivante :

for *nom* **in** *expr* **while** *cond* **do** *instructions* **od**

Le principe est similaire à celui de la boucle **for - from**, mais les valeurs successives données à *nom* sont les opérandes de l'expression *expr*.

Calculons par exemple la somme des éléments positifs d'une liste :

```
> L:=[1,-2,3,-1,5]:S:=0:
  for k in L do if k>0 then S:=S+k fi od:
  S;
```

Fonctions et procédures

Définition d'une procédure

La syntaxe générale d'une procédure est la suivante :

```
> proc(S)  #pas de point-virgule avant local !
  local L;
  global G;
  options O;
  description D;
  C
  end;
```

S est la séquence des paramètres formels reçus par la procédure. Chaque paramètre formel peut être accompagné d'une spécification de type, sous la forme **nom_param::nom_type** Dans ce cas, un message d'erreur sera renvoyé si le paramètre effectif transmis lors de l'appel n'est pas du type convenable. Si la procédure est appelée avec un nombre insuffisant de paramètres, un message d'erreur est retourné ; il est par contre possible de transmettre plus de paramètres qu'il n'est prévu dans **S** : durant l'exécution de la procédure, les mots réservés **nargs** et **args** représentent respectivement le nombre et la séquence des paramètres effectivement transmis lors de l'appel.

L (resp. **G**) est la séquence des noms des variables locales (resp. globales) utilisées dans la procédure. Pour plus de détails sur les options et la description, voir la feuille d'aide de **procedure**.

Les quatre lignes commençant par **local**, **global**, **options**, **description** sont facultatives.

C est une suite d'instructions formant le *corps* de la procédure : ces instructions sont exécutées séquentiellement lors de l'appel de la procédure.

Résultat renvoyé par une procédure

Toute procédure peut être utilisée comme une fonction : le résultat de l'appel est alors la dernière expression évaluée lors de l'exécution.

Utilisons les remarques précédentes pour écrire une fonction MAX renvoyant le plus grand d'un nombre quelconque d'arguments numériques :

```
> MAX:=proc()
  local m,k;
  m:=args[1];
  for k from 2 to nargs do
    if m<args[k] then m:=args[k] fi
  od;
  m; #renvoi du résultat
  end:
> MAX(3,5,1);
```

$$5 \qquad \textbf{(3.2.1)}$$

La commande **return expr** interrompt l'exécution et renvoie la valeur de **expr**.

On peut améliorer la fonction MAX pour renvoyer un résultat non évalué lorsque les arguments ne sont pas tous numériques :

```
> MAX:=proc()
  local m,k;
  if remove(type,[args],numeric)=[] then
    m:=args[1];
    for k from 2 to nargs do
      if m<args[k] then m:=args[k] fi
    od;
    return m
  else return 'MAX'(args)
  fi;
  end:
> MAX(a,0,1);
```

$$MAX(a, 0, 1) \qquad \textbf{(3.2.2)}$$

```
> MAX(4, 2, 1);
```

$$4 \qquad \textbf{(3.2.3)}$$

La commande **error expr** permet d'interrompre l'exécution tout en envoyant un message d'erreur à l'utilisateur : c'est **expr** qui est affichée, en général une chaîne de caractères.

Exemple : **if nargs=0 then error "il faut au moins un argument" fi**.

Affichage du texte d'une procédure

Comme les tableaux, les procédures obéissent à la règle dite "*last name evaluation*". Pour faire afficher le texte d'une procédure, il faut utiliser **eval** ou **print**.

Maple se charge de formater l'affichage (mots-clés en gras, indentation, suppression des points-virgules inutiles...).

Maple 12 - Calcul numérique

Calculs avec des entiers

Maple peut être utilisé comme un puissant calculateur.

```
> 32*12^13;
```

Ne pas oublier le ";", ni de taper [Entrée].

$$3423782572130304 \quad \textbf{(1.1)}$$

Outre les opérateurs usuels +,-,*,/, Maple reconnait les opérateurs arithmétiques suivants : factorielle (!), puissance (^ ou **), quotient de division entière (*iquo(.,.)*), reste de division entière (*irem(.,.)*).

```
> 100!;
```

Le dernier résultat évalué (dans le temps) par Maple est affecté au caractère % (*dito*).
ifactor(n) retourne la décomposition de *n* en facteurs premiers.

```
> ifactor(%);
```

$$(2)^{97}\,(3)^{48}\,(5)^{24}\,(7)^{16}\,(11)^{9}\,(13)^{7}\,(17)^{5}\,(19)^{5}\,(23)^{4}\,(29)^{3}\,(31)^{3}\,(37)^{2}\,(41)^{2}\,(43)^{2}\,(47)^{2}\,(53)\,(59)\,(61)\,(67)\,(71)\,(73)\,(79)\,(83)\,(89)\,(97) \quad \textbf{(1.2)}$$

Calculs avec des réels

L'un des intérêts principaux de Maple réside en son aptitude à effectuer des calculs exacts.

Fractions et radicaux ne sont pas immédiatement convertis en leur approximation décimale, ce qui évite les erreurs d'arrondi. Considérer l'expression $\frac{2^{30}\sqrt{3}}{3^{20}}$.

```
> a:=(2^30/3^20)*sqrt(3);
```

L'affectation se fait par le "deux points, égale".

$$a := \frac{1073741824}{3486784401}\sqrt{3} \quad \textbf{(2.1)}$$

Maple peut aussi en donner une approximation décimale (en virgule flottante) avec la commande **evalf** .

```
> evalf(a);
```

Le nombre de chiffres significatifs retournés par *evalf* est déterminé par la variable *Digits* qui par défaut vaut 10.

$$0.5333783739 \quad \textbf{(2.2)}$$

```
> Digits:=20;evalf(a);evalf(a,15);
```

$$Digits := 20$$

$$0.53337837373779145537$$

$$0.533378373737793 \quad \textbf{(2.3)}$$

Maple peut calculer des sommes finies ou infinies. Soit la somme finie $\sum_{i=1}^{10}\frac{1+i}{1+i^4}$.

Dans les identificateurs, Maple fait la distinction entre les majuscules et les minuscules. Par exemple, **Sum** est la forme inerte de la fonction **sum**. **Sum** est reconnu par le programme d'affichage qui retourne une "belle" somme et par la fonction **value** qui le transforme en **sum** et essaie de faire le calcul de la somme.

```
S:=Sum((1+i)/(1+i^4),i=1..10);
S=value(S); #ceci est une égalité et non une affectation
sum((1+i)/(1+i^4),i=1..10);
```

$$S := \sum_{i=1}^{10}\frac{1+i}{1+i^4}$$

$$\sum_{i=1}^{10}\frac{1+i}{1+i^4} = \frac{51508056727594732913722}{40626648938819200088497}$$

$$\frac{51508056727594732913722}{40626648938819200088497} \quad \textbf{(2.4)}$$

```
> Sum(1/k^2,k=1..infinity);
```

$$\sum_{k=1}^{\infty} \frac{1}{k^2} \tag{2.5}$$

```
> value(%);
```

$$\frac{1}{6}\,\pi^2 \tag{2.6}$$

Il est possible d'obtenir les valeurs des fonctions élémentaires et de nombreuses fonctions spéciales ou constantes. Par exemple, une valeur approchée de la base e des logarithmes népériens à 40 décimales.

```
> evalf(exp(1),40);
```

2.7182818284590452353602874713526624977572 **(2.7)**

Enfin une évaluation de π à 500 décimales.

```
> evalf(Pi,500);
```

Calculs avec les nombres complexes

Maple peut aussi effectuer des calculs avec des nombres complexes. Le nombre complexe *i=sqrt(-1)* est représenté par la lettre majuscule **I**.

```
> (3+5*I)/(7+4*I);
```

$$\frac{41}{65} + \frac{23}{65}\,\mathrm{I} \tag{3.1}$$

Il est facile d'obtenir la forme trigonométrique d'un nombre complexe à l'aide de la fonction **convert**. Maple utilise la forme polaire (*r*, θ) où *r* est le module et θ est l'argument du complexe considéré.

```
> convert(%, polar);
```

$$\mathrm{polar}\left(\frac{1}{65}\sqrt{2210}, \arctan\left(\frac{23}{41}\right)\right) \tag{3.2}$$

Fonctions et constantes reconnues par Maple

Fonctions élémentaires

floor(x) : partie entière "mathématique" de *x*
trunc(x), frac(x) : parties entière et fractionnaire "informatiques" de *x*
round(x)
sqrt(x), abs(x)
exp(x), ln(x), log10(x), log[b](x)
sin(x), cos(x), tan(x), cot(x), arcsin(x), arccos(x), arctan(x), arccot(x)
sinh(x), cosh(x), tanh(x), cotanh(x), arcsinh(x), arccosh(x), arctanh(x)

Constantes

Pi : 3,14159... ; **I** : racine de -1 ; **infinity** : + l'infini ; **gamma** : la constante d'Euler...

Dans Maple 12, un certain nombre de constantes sont accessibles via la palette "Constants and Symbols", mais pour taper **Pi** au clavier, bien distinguer majuscule/minuscule !

```
> evalf(pi),evalf(Pi),evalf(PI);
```

π, 3.141592654, Π **(4.2.1)**

Calcul formel

Affectation et évaluation

Maple permet de travailler sur des objets tels que des valeurs numériques, des fonctions, des procédures, des listes ou des tableaux. Mais Maple permet aussi de calculer formellement avec des expressions contenant des variables *libres* c'est à dire des variables auxquelles on n'a affecté aucune valeur particulière.

Affectation

Elle se fait par le "deux points, égale" (**:=**)

Variables libres et évaluation

restart permet de réinitialiser toutes les variables.

```
> P:=x^2+x+1;
```

$$P := x^2 + x + 1 \quad \textbf{(5.2.1)}$$

Maple sait que x est une variable libre et l'évaluation qu'il fait de $x^2 + x + 1$ est une expression en x qu'il attribue à P.

```
> x:=1;P;
```

$$x := 1 \quad \textbf{(5.2.2)}$$

$$3$$

```
> x:=2;P;
```

$$x := 2 \quad \textbf{(5.2.3)}$$

$$7$$

```
> Q:=x^4-3;
```

$$Q := 13 \quad \textbf{(5.2.4)}$$

C'est la valeur de $x^4 - 3$ pour $x = 2$ qui est affectée à Q, ce n'est donc pas une expression en x, mais une valeur numérique.

```
> x:=1;Q;
```

$$x := 1 \quad \textbf{(5.2.5)}$$

$$13$$

Pour pouvoir retrouver l'expression de P en fonction de x et construire de nouvelles expressions en x, on peut de nouveau transformer x en variable libre en lui affectant son nom : on met ce dernier entre apostrophes.

```
> x:='x';P;Q;
```

$$x := x \quad \textbf{(5.2.6)}$$

$$x^2 + x + 1$$

$$13$$

Pour Maple, une variable libre (non affectée) est une variable à laquelle est affecté son nom.

Développements, factorisation et simplifications d'expressions

La fonction *expand*

La fonction **expand** permet

- de développer les expressions polynomiales
- d'exprimer les lignes trigonométriques de $n.x$ en fonction de celles de x.

```
> expand((x+y)^7);
```

$$x^7 + 7\,x^6\,y + 21\,x^5\,y^2 + 35\,x^4\,y^3 + 35\,x^3\,y^4 + 21\,x^2\,y^5 + 7\,x\,y^6 + y^7 \quad \textbf{(6.1.1)}$$

```
> expand(sin(4*x));
```

$$8\,\sin(x)\,\cos(x)^3 - 4\,\sin(x)\,\cos(x) \quad \textbf{(6.1.2)}$$

La fonction *factor*

La fonction **factor** permet de factoriser une expression polynomiale d'une ou plusieurs variables ou un quotient de telles expressions.

```
> factor(x^4+x^2+1);
```

$$(x^2 + x + 1)\,(x^2 - x + 1) \quad \textbf{(6.2.1)}$$

Plus précisément, **factor** factorise sur le sous-corps des complexes engendré par les coefficients :

```
> factor(x^2-2);factor(sqrt(2)*(x^2-2));
```

$$x^2 - 2$$

$$-\sqrt{2}\,\left(-x + \sqrt{2}\right)\,\left(x + \sqrt{2}\right) \quad \textbf{(6.2.2)}$$

La fonction *normal*

La fonction **normal** réduit au même dénominateur et simplifie les expressions rationnelles telles que $\frac{x^3 - y^3}{x^2 + x - y - y^2}$ ou $\frac{\sin(x)^3 - \cos(x)^3}{\sin(x) - \cos(x)}$.
Elle divise par le PGCD mais ne factorise pas.

```
> normal((x^3-y^3)/(x^2+x-y-y^2));
```

$$\frac{x^2 + x\,y + y^2}{x + 1 + y} \quad \textbf{(6.3.1)}$$

À la différence de **factor**, la fonction **normal** ne peut simplifier que si les coefficients du numérateur et du dénominateur sont rationnels.

```
> f:=(x^2-3)/(x-sqrt(3)):normal(f),factor(f);
```

$$-\frac{x^2 - 3}{-x + \sqrt{3}},\ x + \sqrt{3} \quad \textbf{(6.3.2)}$$

La fonction *simplify*

Soit $\sin(x)^4 + 2\,\cos(x)^2 - 2\,\sin(x)^2 - \cos(2\,x)$. Maple peut utiliser des identités remarquables pour simplifier une telle expression.

```
> simplify(sin(x)^4+2*cos(x)^2-2*sin(x)^2-cos(2*x));
```

$$\cos(x)^4 \quad \textbf{(6.4.1)}$$

La fonction *convert*

Utilisée avec l'option **parfrac**, elle permet de décomposer une fraction rationnelle en éléments simples.

```
> restart:F:=(a*x^2+b)/(x*(-3*x^2-x+4));convert( F,parfrac,x);
```

$$F := \frac{a\,x^2 + b}{x\,(-3\,x^2 - x + 4)}$$

$$\frac{1}{7}\,\frac{-a - b}{x - 1} + \frac{1}{4}\,\frac{b}{x} + \frac{1}{28}\,\frac{-16\,a - 9\,b}{3\,x + 4} \quad \textbf{(6.5.1)}$$

Transformation d'une expression trigonométrique en une expression exponentielle.

```
> convert(cot(x),exp);
```

$$\frac{I\,(e^{I\,x} + e^{-I\,x})}{e^{I\,x} - e^{-I\,x}} \quad \textbf{(6.5.2)}$$

Fonctions d'une variable

Il existe plusieurs manières de définir une fonction. L'une d'elles utilise l'opérateur **->** (opérateur **arrow**) avec une syntaxe voisine de celle couramment employée en mathématiques.

Soit la fonction $x \to x^2 + \frac{1}{2}$.

```
> f:=x->x^2+1/2;
```

$$f := x \to x^2 + \frac{1}{2} \tag{7.1}$$

Contrairement aux expressions, pour définir une fonction il n'est pas nécessaire que la variable soit libre.

```
> f(2),f(a+b);
```

$$\frac{9}{2},\ (a+b)^2 + \frac{1}{2} \tag{7.2}$$

Si P est une expression dépendant d'une variable x, pour obtenir sa valeur pour $x = a$, on peut :
soit affecter a à x, puis demander l'évaluation de P (x n'est plus alors une variable libre),
soit utiliser **subs(x=a,P)** qui retourne le résultat obtenu en substituant a à x dans l'expression de P (dans ce cas x reste une variable libre) :

```
> P:=x^2+1/2;subs(x=2,P),subs(x=a+b,P);
```

$$P := x^2 + \frac{1}{2}$$

$$\frac{9}{2},\ (a+b)^2 + \frac{1}{2} \tag{7.3}$$

Passerelles entre expressions et fonctions

Si P est une expression de la variable libre x, l'évaluation de **unapply(P,x)** retourne la fonction qui à x associe P. Idem avec plusieurs variables.

```
> x:='x':P:=x^3-x+1;g:=unapply(P,x);
```

$$P := x^3 - x + 1$$

$$g := x \to x^3 - x + 1 \tag{7.4}$$

Penser à utiliser **unapply** lorsque Maple renvoie un résultat sous forme d'expression.

```
> dsolve(diff(y(x),x)=2*y(x)-4,y(x));
```

$$y(x) = 2 + e^{2x}\_C1 \tag{7.5}$$

```
> f:=unapply(rhs(%),_C1,x);
```

$$f := (\_C1, x) \to 2 + e^{2x}\_C1 \tag{7.6}$$

rhs renvoie le membre de droite d'une égalité ("right hand side", *vs* **lhs**...).
Inversement, si f est une fonction, l'appel de f (avec le bon nombre d'arguments !) renvoie une expression.

```
> f(3,t);
```

$$2 + 3\,e^{2t} \tag{7.7}$$

Résolutions d'équations

Maple peut résoudre des équations algébriques comme $x^3 - \frac{a\,x^2}{2} + \frac{13\,x^2}{3} = \frac{13\,a\,x}{6} + \frac{10\,x}{3} - \frac{5\,a}{3}$.

```
> eqn:= x^3-1/2*a*x^2+13/3*x^2=13/6*a*x+10/3*x-5/3*a;
```

$$eqn := x^3 - \frac{1}{2}\,a\,x^2 + \frac{13}{3}\,x^2 = \frac{13}{6}\,a\,x + \frac{10}{3}\,x - \frac{5}{3}\,a \tag{8.1}$$

```
> solve(eqn,x);
```

$$-5,\ \frac{2}{3},\ \frac{1}{2}\,a \tag{8.2}$$

Pour un système, on remplace l'équation par un ensemble d'équations et l'inconnue par l'ensemble des inconnues :

```
> syst:={x+y=a,x-y=b};
```

$$syst := \{x - y = b, x + y = a\} \quad \textbf{(8.3)}$$

```
> solve(syst,{x,y});
```

$$\left\{x = \frac{1}{2}\ b + \frac{1}{2}\ a,\ y = -\frac{1}{2}\ b + \frac{1}{2}\ a\right\} \quad \textbf{(8.4)}$$

assign permet d'affecter les solutions aux noms des inconnues (qui ne seront alors plus des variables libres ...) en vue d'une utilisation ultérieure.

```
> assign(%);x,y;
```

$$\frac{1}{2}\ b + \frac{1}{2}\ a,\ -\frac{1}{2}\ b + \frac{1}{2}\ a \quad \textbf{(8.5)}$$

Il est aussi possible de résoudre des équations comprenant des expressions trigonométriques.

```
> x:='x': solve(arccos(x)=arctan(x),x);
```

$$\frac{1}{2}\sqrt{-2 + 2\sqrt{5}} \quad \textbf{(8.6)}$$

```
> solve(cos(x)=0,x); #Maple ne donne pas toujours toutes les solutions...
```

$$\frac{1}{2}\ \pi \quad \textbf{(8.7)}$$

RootOf ; allvalues

Lorsqu'il ne trouve pas *a priori* leurs valeurs exactes, Maple exprime les racines d'un polynôme à l'aide de **RootOf** ; **allvalues** permet de forcer un calcul explicite : Maple trouve alors parfois les valeurs exactes, sinon il fournit des valeurs approchées.

```
> solve(x^4-2*x^3+x^2-2*x+1=0,x);
```

$$RootOf(\_Z^4 - 2\ \_Z^3 + \_Z^2 - 2\ \_Z + 1,\ index = 1),\ RootOf(\_Z^4 - 2\ \_Z^3 + \_Z^2 - 2\ \_Z + 1,\ index = 2),\ RootOf(\_Z^4 - 2\ \_Z^3 + \_Z^2 - 2\ \_Z + 1,\ index = 3),\ RootOf(\_Z^4 - 2\ \_Z^3 + \_Z^2 - 2\ \_Z + 1,\ index = 4) \quad \textbf{(8.8)}$$

```
> map(allvalues,[%]);
```

$$\left[\frac{1}{2}\sqrt{2} + \frac{1}{2} - \frac{1}{2}\sqrt{-1 + 2\sqrt{2}},\ \frac{1}{2}\sqrt{2} + \frac{1}{2} + \frac{1}{2}\sqrt{-1 + 2\sqrt{2}},\ \frac{1}{2} - \frac{1}{2}\sqrt{2} + \frac{1}{2}\sqrt{-1 - 2\sqrt{2}},\ \frac{1}{2} - \frac{1}{2}\sqrt{2} - \frac{1}{2}\sqrt{-1 - 2\sqrt{2}}\right] \quad \textbf{(8.9)}$$

```
> solve(z^5+z^3+1=0,z);
```

$$RootOf(\_Z^5 + \_Z^3 + 1,\ index = 1),\ RootOf(\_Z^5 + \_Z^3 + 1,\ index = 2),\ RootOf(\_Z^5 + \_Z^3 + 1,\ index = 3),\ RootOf(\_Z^5 + \_Z^3 + 1,\ index = 4),\ RootOf(\_Z^5 + \_Z^3 + 1,\ index = 5) \quad \textbf{(8.10)}$$

```
> map(allvalues,[%]);
```

$$[RootOf(\_Z^5 + \_Z^3 + 1,\ index = 1),\ RootOf(\_Z^5 + \_Z^3 + 1,\ index = 2),\ RootOf(\_Z^5 + \_Z^3 + 1,\ index = 3),\ RootOf(\_Z^5 + \_Z^3 + 1,\ index = 4),\ RootOf(\_Z^5 + \_Z^3 + 1,\ index = 5)] \quad \textbf{(8.11)}$$

fsolve donne directement des valeurs approchées des solutions (uniquement des solutions réelles, par défaut : penser à utiliser l'option **complex**).

```
> fsolve(z^5+z^3+1=0,z);
```

$$-0.8376197748 \quad \textbf{(8.12)}$$

```
> fsolve(z^5+z^3+1=0,z,complex);
```

$$-0.8376197748,\ -0.2178532194 - 1.166951246\ I,\ -0.2178532194 + 1.166951246\ I,\ 0.6366631068 - 0.6647015651\ I,\ 0.6366631068 + 0.6647015651\ I \quad \textbf{(8.13)}$$

Maple 12 - Algèbre linéaire

Maple 12 dispose d'une bibliothèque *(package)* **LinearAlgebra** qui offre de nombreux outils pour créer et manipuler vecteurs et matrices. Cette bibliothèque doit être chargée en mémoire :

```
> restart;with(LinearAlgebra);
```

[*&x, Add, Adjoint, BackwardSubstitute, BandMatrix, Basis, BezoutMatrix, BidiagonalForm, BilinearForm, CharacteristicMatrix, CharacteristicPolynomial, Column, ColumnDimension, ColumnOperation, ColumnSpace, CompanionMatrix, ConditionNumber, ConstantMatrix, ConstantVector, Copy, CreatePermutation, CrossProduct, DeleteColumn, DeleteRow, Determinant, Diagonal, DiagonalMatrix, Dimension, Dimensions, DotProduct, EigenConditionNumbers, Eigenvalues, Eigenvectors, Equal, ForwardSubstitute, FrobeniusForm, GaussianElimination, GenerateEquations, GenerateMatrix, Generic, GetResultDataType, GetResultShape, GivensRotationMatrix, GramSchmidt, HankelMatrix, HermiteForm, HermitianTranspose, HessenbergForm, HilbertMatrix, HouseholderMatrix, IdentityMatrix, IntersectionBasis, IsDefinite, IsOrthogonal, IsSimilar, IsUnitary, JordanBlockMatrix, JordanForm, KroneckerProduct, LA_Main, LUDecomposition, LeastSquares, LinearSolve, Map, Map2, MatrixAdd, MatrixExponential, MatrixFunction, MatrixInverse, MatrixMatrixMultiply, MatrixNorm, MatrixPower, MatrixScalarMultiply, MatrixVectorMultiply, MinimalPolynomial, Minor, Modular, Multiply, NoUserValue, Norm, Normalize, NullSpace, OuterProductMatrix, Permanent, Pivot, PopovForm, QRDecomposition, RandomMatrix, RandomVector, Rank, RationalCanonicalForm, ReducedRowEchelonForm, Row, RowDimension, RowOperation, RowSpace, ScalarMatrix, ScalarMultiply, ScalarVector, SchurForm, SingularValues, SmithForm, StronglyConnectedBlocks, SubMatrix, SubVector, SumBasis, SylvesterMatrix, ToeplitzMatrix, Trace, Transpose, TridiagonalForm, UnitVector, VandermondeMatrix, VectorAdd, VectorAngle, VectorMatrixMultiply, VectorNorm, VectorScalarMultiply, ZeroMatrix, ZeroVector, Zip*] **(1)**

Pour ne pas afficher la (longue) liste des fonctions disponibles, il suffit de remplacer "**;**" par "**:**".

Vecteurs et matrices

Création de vecteurs et de matrices

C'est la fonction **Vector** qui permet de définir des vecteurs.
On peut donner simplement la *liste (séquence entre crochets)* des composantes :

```
> C:=Vector([1,2,3]);
```

$$C := \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \quad \textbf{(1.1.1)}$$

Par défaut, on obtient un "vecteur colonne". Pour obtenir une ligne, ajouter l'option [row] avant les paramètres :

```
> L := Vector[row]([1, 2, 3]);
```

$$L := \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \quad \textbf{(1.1.2)}$$

Maple 12 accepte également les notations suivantes :

```
> C := ⟨1, 2, 3⟩; L := ⟨1|2|3⟩;
```

On peut déclarer la dimension et la liste (éventuellement vide) des premières composantes, ou encore un "symbole" pour engendrer un vecteur "générique" :

```
> v:=Vector[row](4,[1,1]);x:=Vector[row](3,symbol=a);
```

$$v := \begin{bmatrix} 1 & 1 & 0 & 0 \end{bmatrix}$$

$$x := \begin{bmatrix} a_1 & a_2 & a_3 \end{bmatrix} \quad \textbf{(1.1.3)}$$

On peut donner la dimension et une fonction associant chaque composante à son rang :

```
> v:=Vector(5,k->k^2);
```

C'est la fonction **Matrix** qui permet de définir des matrices, avec des syntaxes similaires :

```
> Matrix([[1,2,3],[3,2,1]]);Matrix(3,2,[1,2,3]);Matrix(2,3,(i,j)->1/(i+j-1));
```

Voir aussi **IdentityMatrix**, **DiagonalMatrix**, **BandMatrix**, **VandermondeMatrix**, **JordanBlockMatrix**.

Maple 12 permet enfin de saisir des matrices dans l'interface graphique via la "palette" Matrix.

Accès aux éléments

Pour obtenir la valeur d'une composante d'un vecteur ou d'une matrice, il suffit d'indiquer sa ou ses "coordonnées" entre crochets :

```
> v[4];m[2,1];
```

De même, pour modifier un élément, il suffit d'une affectation :

```
> v[4]:=2;m[2,1]:=a;
```

Voir aussi **Row**, **Column**, **SubMatrix** pour extraire des "morceaux" d'une matrice.

Pour obtenir la dimension d'un vecteur ou les dimensions d'une matrice, utiliser **Dimension**, **RowDimension**, **ColumnDimension**.

Premières opérations

Combinaisons linéaires de vecteurs et de matrices

On peut écrire les combinaisons linéaires de façon naturelle avec **+** et *****.

```
> u:=<1|1|1>;v:=<1|2|3>;w:=2*u+3*v;
```

$$u := \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$$

$$v := \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

$$w := \begin{bmatrix} 5 & 8 & 11 \end{bmatrix} \quad \textbf{(2.1.1)}$$

Produit matriciel

***Il ne faut pas utiliser** ** avec des matrices ou des vecteurs, car Maple envoie un message d'erreur.

```
> A:=Matrix(3,3,(i,j)->i+j-1);B:=Matrix(3,3,1);
```

$$A := \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix}$$

$$B := \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad \textbf{(2.2.1)}$$

```
> A*B;
Error, (in rtable/Product) invalid arguments
```

On peut utiliser le point ("dot") ou la fonction **Multiply** mais cette dernière ne s'applique qu'à deux matrices :

```
> A.B,Multiply(A,B);
```

$$\begin{bmatrix} 6 & 6 & 6 \\ 9 & 9 & 9 \\ 12 & 12 & 12 \end{bmatrix}, \begin{bmatrix} 6 & 6 & 6 \\ 9 & 9 & 9 \\ 12 & 12 & 12 \end{bmatrix} \quad \textbf{(2.2.2)}$$

Un objet de type **Vector** se comporte comme une matrice de dimensions correspondant au type de vecteur :

```
> v:=<1,2,3>;B.v;v.B;
```

$$v := \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

$$\begin{bmatrix} 6 \\ 6 \\ 6 \end{bmatrix}$$

```
Error, (in LinearAlgebra:-VectorMatrixMultiply) invalid input:
LinearAlgebra:-VectorMatrixMultiply expects its 1st argument, v, to be of
type Vector[row] but received Vector(3, {(1) = 1, (2) = 2, (3) = 3})
```

Dans une combinaison linéaire de matrices carrées d'ordre n, une valeur numérique ou une variable libre x est remplacée par la matrice scalaire $x.I_n$, à condition d'évaluer l'expression avec **evalm** :

```
> Matrix(3,3,1)-x,evalm(Matrix(3,3,1)-x);
```

$$-x+\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, \begin{bmatrix} 1-x & 1 & 1 \\ 1 & 1-x & 1 \\ 1 & 1 & 1-x \end{bmatrix} \qquad \textbf{(2.2.3)}$$

Puissances, inverse

Utiliser l'opérateur **^** (ou ******) ; si la matrice est inversible, on peut calculer les puissances négatives :

```
> A:=Matrix([[a,b],[-b,a]]);A^2;A^(-1);
```

$$A := \begin{bmatrix} a & b \\ -b & a \end{bmatrix}$$

$$\begin{bmatrix} a^2-b^2 & 2\,a\,b \\ -2\,a\,b & a^2-b^2 \end{bmatrix}$$

$$\begin{bmatrix} \frac{a}{a^2+b^2} & -\frac{b}{a^2+b^2} \\ \frac{b}{a^2+b^2} & \frac{a}{a^2+b^2} \end{bmatrix} \qquad \textbf{(2.3.1)}$$

La fonction **MatrixInverse** renvoie également la matrice inverse.

Polynômes de matrices

Pour évaluer $P(A)$, où P est un polynôme et A une matrice carrée, on peut définir P comme une **fonction** ou comme une **expression** :

```
> A:=DiagonalMatrix([1,1,-1]);
```

$$A := \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \qquad \textbf{(2.4.1)}$$

```
> P := x -> x^2-1; P(A);
```

$$P := x \to x^2-1$$

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \textbf{(2.4.2)}$$

```
> P:=X^2-1;subs(X=A,P);evalm(%);
```

$$P := X^2-1$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}^2-1$$

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \textbf{(2.4.3)}$$

Recherche de bases

Sous-espace défini par des générateurs

La fonction **Basis**, appliquée à une liste (*resp.* un ensemble) de vecteurs de même dimension, renvoie une base du sous-espace qu'ils engendrent, sous forme de liste (*resp.* d'ensemble). Pour obtenir la dimension de ce sous-espace, il suffit d'appliquer au résultat la fonction **nops** qui renvoie le nombre d'éléments de la base obtenue.
Voir aussi **ColumnSpace** et **RowSpace** pour obtenir une base du sous-espace engendré par les colonnes ou les lignes d'une matrice.

Intersection et somme de sous-espaces

Appliquer **IntersectionBasis** ou **SumBasis** à une séquence de listes de vecteurs, représentant chacune un sous-espace :

```
> E1:=[<1,2,0>,<2,1,3>]:E2:=[<1,2,1>,<-1,1,1>,<2,1,0>]:
  IntersectionBasis([E1,E2]),SumBasis([E1,E2]);
```

$$\left[\begin{bmatrix} -5 \\ -7 \\ -3 \end{bmatrix}\right], \left[\begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix}, \begin{bmatrix} 2 \\ 1 \\ 3 \end{bmatrix}, \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}\right] \qquad \textbf{(3.2.1)}$$

Noyau et rang d'une matrice

La fonction **NullSpace**, appliquée à une matrice, retourne une base du noyau de l'application linéaire associée ; la fonction **Rank** permet d'obtenir le rang d'une matrice.

Précautions à prendre en cas de coefficients formels

Attention aux calculs "masqués" effectués par Maple, qui n'envisage pas les cas particuliers liés à des valeurs pouvant être prises par des variables libres (par exemple, pour Maple, **x** n'est pas 0, ni 1, etc. (voir les rangs ci-dessous !).
De plus, **eval** ne suffit pas toujours à l'évaluation complète ; on peut utiliser **map** pour évaluer chaque coefficient :

```
> A:=VandermondeMatrix([x,y,z]);
  x:=1;y:=1;eval(A),map(eval,A),Rank(A),Rank(map(eval,A));
```

$$A := \begin{bmatrix} 1 & x & x^2 \\ 1 & y & y^2 \\ 1 & z & z^2 \end{bmatrix}$$

$$x := 1$$

$$y := 1$$

$$\begin{bmatrix} 1 & x & x^2 \\ 1 & y & y^2 \\ 1 & z & z^2 \end{bmatrix}, \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & z & z^2 \end{bmatrix}, 3, 2 \qquad \textbf{(3.4.1)}$$

Résolution de systèmes linéaires

Système donné matriciellement

Si A est une matrice à n lignes et p colonnes et B un vecteur à n composantes (*resp.* une matrice à n lignes et q colonnes), **LinearSolve(A,B)** renvoie la solution générale du système $AX = B$. Dans le cas où le système admet une infinité de solutions, Maple les exprime à l'aide de paramètres $\_t_1$, $\_t_2$, ... :

```
> A:=Matrix(3,3,1);B:=Vector(3,1);
  LinearSolve(A,B);
```

$$A := \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

$$B := \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 - \_t0_2 - \_t0_3 \\ \_t0_2 \\ \_t0_3 \end{bmatrix} \quad \textbf{(4.1.1)}$$

Système donné par des équations

Si l'on utilise la fonction **solve** pour résoudre un système d'équations linéaires, on n'obtient pas directement un vecteur. Cela dit, la forme du résultat permet l'utilisation de **subs** et **Vector** pour construire le vecteur souhaité :

```
> eqs:={2*x+3*y-2*z,x+y+z-1,5*x+4*y+2*z};
  S:=solve(eqs,{x,y,z});
```

$$eqs := \{2x + 3y - 2z, 5x + 4y + 2z, x + y + z - 1\}$$

$$S := \{x = -2, y = 2, z = 1\} \quad \textbf{(4.2.1)}$$

```
> v:=Vector(subs(S,[x,y,z]));
```

$$v := \begin{bmatrix} -2 \\ 2 \\ 1 \end{bmatrix} \quad \textbf{(4.2.2)}$$

On peut aussi utiliser la commande **assign**, mais elle a l'inconvénient d'affecter les valeurs aux variables, qui ne sont donc plus libres.
À partir d'un système d'équations linéaire, on peut également recourir à la fonction **LinearSolve** en construisant, à l'aide de **GenerateMatrix**, la matrice du système et le vecteur associé au second membre :

```
> GenerateMatrix(eqs,[x,y,z]);
```

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 3 & -2 \\ 5 & 4 & 2 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \quad \textbf{(4.2.3)}$$

```
> LinearSolve(%);
```

$$\begin{bmatrix} -2 \\ 2 \\ 1 \end{bmatrix} \quad \textbf{(4.2.4)}$$

Déterminants

C'est la fonction **Determinant** qui permet le calcul du déterminant d'une matrice carrée :

```
> A:=Matrix(3,3,1)+a;Determinant(A);
```

$$A := a + \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

$$3a^2 + a^3 \quad \textbf{(5.1)}$$

```
> A:=VandermondeMatrix([x,y,z]);Determinant(A);factor(%);
```

$$A := \begin{bmatrix} 1 & x & x^2 \\ 1 & y & y^2 \\ 1 & z & z^2 \end{bmatrix}$$

$$y z^2 - y^2 z + z x^2 - x z^2 + x y^2 - y x^2$$

$$-(-z+y)\ (x-z)\ (x-y) \qquad \textbf{(5.2)}$$

Polynôme caractéristique d'une matrice

Si A est une matrice carrée d'ordre n, l'évaluation de **CharacteristicPolynomial(A,x)** retourne le déterminant de la matrice $xI_n - A$:

```
> A:=Matrix(3,3,(i,j)->i^j);
```

$$A := \begin{bmatrix} 1 & 1 & 1 \\ 2 & 4 & 8 \\ 3 & 9 & 27 \end{bmatrix} \qquad \textbf{(5.1.1)}$$

```
> p:=x->CharacteristicPolynomial(A,x):p(x); # On peut vérifier ...
```

$$-12 + x^3 - 32 x^2 + 62 x \qquad \textbf{(5.1.2)}$$

```
> p(A);      # ... le théorème de Cayley-Hamilton !
```

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \textbf{(5.1.3)}$$

Valeurs propres et vecteurs propres

Polynôme caractéristique, polynôme minimal

Les fonctions **CharacteristicPolynomial et MinimalPolynomial** de la bibliothèque **LinearAlgebra** renvoient respectivement le polynôme caractéristique et le polynôme minimal d'une matrice carrée ; le second paramètre spécifie l'indéterminée souhaitée :

```
> M:=Matrix(3,3,1)+a;
  factor(CharacteristicPolynomial(M,lambda));factor(MinimalPolynomial(M,X));
```

$$M := a + \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

$$-(a-\lambda+3)\ (a-\lambda)^2$$

$$(X-a)\ (X-3-a) \qquad \textbf{(6.1.1)}$$

Attention ! Le polynôme caractéristique de A, pour Maple, est $x \to det(x.I_n - A)$ donc l'oppposé du nôtre pour n impair.

Valeurs propres

Outre la recherche des racines du polynôme caractéristique, la fonction **Eigenvalues** permet d'obtenir les valeurs propres d'une matrice. Si A est une matrice carrée, **Eigenvalues(A)** retourne, sous forme explicite ou implicite, la séquence des valeurs propres de A.
Lorsque la matrice est d'ordre inférieur ou égal à 4, **Eigenvalues(A)** retourne la séquence des valeurs propres de A avec une écriture utilisant des radicaux.

```
> M:=Matrix(3,3,[sqrt(2),-1,1,2,1,-1,-2,1,-1]);
  Eigenvalues(M);
```

$$M := \begin{bmatrix} \sqrt{2} & -1 & 1 \\ 2 & 1 & -1 \\ -2 & 1 & -1 \end{bmatrix}$$

$$\begin{bmatrix} 0 \\ \frac{1}{2}\sqrt{2} - \frac{1}{2}\,\mathrm{I}\sqrt{14} \\ \frac{1}{2}\sqrt{2} + \frac{1}{2}\,\mathrm{I}\sqrt{14} \end{bmatrix} \quad \textbf{(6.2.1)}$$

Si un élément de M est exprimé en virgule flottante, alors des valeurs approchées des valeurs propres de M sont données dans le même format. Les coefficients de M ne doivent contenir aucun paramètre.

```
> M:=Matrix(3,3,[1.0,2,3,1,2,3,2,5,6]);
  Eigenvalues(M);
```

$$M := \begin{bmatrix} 1.0 & 2 & 3 \\ 1 & 2 & 3 \\ 2 & 5 & 6 \end{bmatrix}$$

$$\begin{bmatrix} 9.32182538049648457 + 0.\,\mathrm{I} \\ -9.59852072122918994\,10^{-16} + 0.\,\mathrm{I} \\ -0.321825380496477464 + 0.\,\mathrm{I} \end{bmatrix} \quad \textbf{(6.2.2)}$$

Vecteurs propres

C'est la fonction **Eigenvectors** qui permet d'obtenir les vecteurs propres d'une matrice.

Si **A** est une matrice carrée, **Eigenvectors(A)** retourne une séquence formée d'un vecteur contenant les valeurs propres (répétées selon leur multiplicité), suivi d'une matrice carrée contenant des vecteurs des sous-espaces propres correspondant. ***Attention !*** Lorsque le sous-espace propre a une dimension strictement inférieure à la multiplicité de la valeur propre, cette dernière matrice carrée n'est pas inversible !!

```
> M := Matrix(3,3,[1,-3,3,3,-5,3,6,-6,4]);
  V,P:=Eigenvectors(M);
```

$$M := \begin{bmatrix} 1 & -3 & 3 \\ 3 & -5 & 3 \\ 6 & -6 & 4 \end{bmatrix}$$

$$V, P := \begin{bmatrix} -2 \\ -2 \\ 4 \end{bmatrix}, \begin{bmatrix} -1 & 1 & \frac{1}{2} \\ 0 & 1 & \frac{1}{2} \\ 1 & 0 & 1 \end{bmatrix} \quad \textbf{(6.3.1)}$$

La matrice est donc diagonalisable :

```
> P^(-1).M.P;
```

$$\begin{bmatrix} -2 & 0 & 0 \\ 0 & -2 & 0 \\ 0 & 0 & 4 \end{bmatrix} \quad \textbf{(6.3.2)}$$

Variante : **Eigenvectors(A,output=list)** retourne une séquence de listes de la forme $\left[\lambda, m, B_\lambda\right]$, où λ est une valeur propre de **A**, m son ordre de multiplicité et B_λ une base du sous-espace propre associé à λ.

```
> Eigenvectors(M,output=list);
```

$$\left[\left[4, 1, \left\{\begin{bmatrix} \frac{1}{2} \\ \frac{1}{2} \\ 1 \end{bmatrix}\right\}\right], \left[-2, 2, \left\{\begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}\right\}\right]\right] \quad \textbf{(6.3.3)}$$

Voir aussi la fonction **JordanForm** :

```
> M:=Matrix(4,4,(i,j)->irem(i+j-2,4));
  T,P:=JordanForm(M,output=['J','Q']); # P contiendra la matrice de passage
```

$$M := \begin{bmatrix} 0 & 1 & 2 & 3 \\ 1 & 2 & 3 & 0 \\ 2 & 3 & 0 & 1 \\ 3 & 0 & 1 & 2 \end{bmatrix}$$

$$T, P := \begin{bmatrix} -2 & 0 & 0 & 0 \\ 0 & 6 & 0 & 0 \\ 0 & 0 & -2\sqrt{2} & 0 \\ 0 & 0 & 0 & 2\sqrt{2} \end{bmatrix}, \begin{bmatrix} \frac{1}{4} & \frac{1}{4} & \frac{1}{8} & \frac{2+3\sqrt{2}}{-1+2\sqrt{2}} & \frac{1}{8}\left(\sqrt{2}-1\right)\sqrt{2} \\ -\frac{1}{4} & \frac{1}{4} & -\frac{1}{8} & \frac{-4+\sqrt{2}}{-1+2\sqrt{2}} & -\frac{1}{8}\sqrt{2} \\ \frac{1}{4} & \frac{1}{4} & -\frac{1}{8} & \frac{2+3\sqrt{2}}{-1+2\sqrt{2}} & -\frac{1}{8}\left(\sqrt{2}-1\right)\sqrt{2} \\ -\frac{1}{4} & \frac{1}{4} & \frac{1}{8} & \frac{-4+\sqrt{2}}{-1+2\sqrt{2}} & \frac{1}{8}\sqrt{2} \end{bmatrix} \quad \textbf{(6.3.4)}$$

Vérification :

```
> P^(-1).M.P;
```

Si le résultat paraît compliqué, utiliser la fonction **simplify**...
Si un élément de M est exprimé en virgule flottante, on obtient un résultat approché en virgule flottante.

Remarque importante

Si la matrice A étudiée dépend d'un ou plusieurs paramètres formels, il est recommandé d'utiliser une ligne de la forme :

```
> T,P:=JordanForm(A,output=['J','Q']);factor(Determinant(P));
```

Si la matrice P est bien définie et inversible, **alors** il est certain que $P^{-1} A P$ est la forme réduite renvoyée par la fonction **JordanForm**.

Mais si, pour certaines valeurs du ou des paramètres, P n'a pas de sens ou n'est pas inversible, alors intervient une **discussion**.

Exemple de matrice non diagonalisable

```
> M:=Matrix(3,3,[2,-3,-1,1,-2,-1,-2,6,3]);
  Eigenvectors(M);
```

$$M := \begin{bmatrix} 2 & -3 & -1 \\ 1 & -2 & -1 \\ -2 & 6 & 3 \end{bmatrix}$$

$$\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 3 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \quad \textbf{(6.4.1)}$$

On peut obtenir une matrice triangulaire supérieure semblable à M.

```
> T,P:=JordanForm(M,output=['J','Q']);
```

$$T, P := \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 2 & 1 \\ 1 & 0 & 0 \\ -2 & 1 & 1 \end{bmatrix} \quad \textbf{(6.4.2)}$$

Mots-clés pour MAPLE

Liste d'opérateurs, fonctions, mots réservés à connaître…

```
^
*
-
+
!
=
<
>
,
;
"
%
$
/
:
&*
()
{}
[]
->
<>
<=
>=
' '
:=
about
abs
add
allvalues
and
arccos, arccosh
arccot, arccoth
arcsin, arcsinh
arctan, arctanh
argument
array
asympt
augment
assume
BandMatrix
binomial
CharacteristicPolynomial
coeff
coeffs
collect
combine
concat
conjugate
convert
cos, cosh
cot, coth
CrossProduct
D
denom
Determinant
DiagonalMatrix
diff
Digits
display
do
DotProduct
dsolve
Eigenvalues
Eigenvectors
eval
evalb
evalf
evalm
exp
expand
factor
false
float
floor
for
frac
fsolve
I
IdentityMatrix
if
ifactor
Im
implicitplot
implicitplot3d
infinity
insequence
int
integer
iquo
irem
is
isprime
ithprime
JordanForm
lhs
limit
LinearAlgebra
LinearSolve
ln
local
log
map, Map
map2, Map2
Matrix
MatrixInverse
max
min
MinimalPolynomial
mod
mul
Multiply
nops
norm
normal
not
NULL
NullSpace
numer
op
or
parfrac
Pi
plot
plot3d
plots
pointplot
polynom
print
proc
product
quo
Rank
Re
rem
remember
restart
return
rhs
RootOf
seq
series
simplify
sin, sinh
solve
sort
spacecurve
sum
sqrt
SubMatrix
subs
tan, tanh
taylor
Trace
Transpose
true
type
unapply
Vector
whattype
while
with
```

ÉCOLE POLYTECHNIQUE – ÉCOLE NORMALE SUPÉRIEURE DE CACHAN

ÉCOLE SUPÉRIEURE DE PHYSIQUE ET DE CHIMIE INDUSTRIELLES

CONCOURS D'ADMISSION 2013 — FILIÈRE MP HORS SPÉCIALITÉ INFO

FILIÈRE PC

COMPOSITION D'INFORMATIQUE – B – (XEC)

(Durée : 2 heures)

L'utilisation des calculatrices **n'est pas autorisée** pour cette épreuve.
Le langage de programmation choisi par le candidat doit être spécifié en tête de la copie.

⋆ ⋆ ⋆

Points fixes de fonctions à domaine fini

Dans ce problème, on s'intéresse aux points fixes des fonctions $f\colon E \to E$, où E est un ensemble fini. Le calcul effectif et efficace des points fixes de telles fonctions est un problème récurrent en informatique (transformation d'automates, vérification automatique de programmes, algorithmique des graphes, etc), et admet différentes approches selon la structure de E et les propriétés de f.

On suppose par la suite un entier strictement positif $n > 0$ fixé et rangé dans une constante globale de même nom, et on pose $\mathsf{E}_n = \{0, \ldots, n-1\}$. On représente une fonction $f\colon \mathsf{E}_n \to \mathsf{E}_n$ par un tableau `t` de taille n, autrement dit $f(x) =$`t[x]` pour tout $x = 0, \ldots, n-1$. Ainsi la fonction f_0 qui à $x \in \mathsf{E}_{10}$ associe $2x + 1$ modulo 10 est-elle représentée par le tableau

1	3	5	7	9	1	3	5	7	9
t[0]	t[1]	t[2]	t[3]	t[4]	t[5]	t[6]	t[7]	t[8]	t[9]

.

Les tableaux sont indexés à partir de 0 et la notation `t`$[i]$ est utilisée dans les questions pour désigner l'élément d'indice i du tableau `t`, indépendamment du langage de programmation choisi. Quel que soit le langage utilisé, on suppose qu'il existe une primitive `allouer(n)` pour créer un tableau d'entiers de taille n (le contenu des cases du nouveau tableau est à priori quelconque). On suppose les entiers machines signés, et on suppose que les entiers $-n, -n+1, \ldots, n-1, n$ ne débordent pas de la capacité des entiers machines – en d'autres termes, les entiers machines représentent fidèlement ces entiers. On suppose que les tableaux peuvent être passés en argument – le type de passage de paramètre, par valeur ou par adresse, devra être précisé par le candidat si le comportement du code écrit venait à en dépendre. On note dans l'énoncé `vrai` et `faux` les deux valeurs possibles d'un booléen. Le candidat reste libre d'utiliser d'autres notations ou d'autres primitives, pourvu qu'elles existent dans le langage de son choix et qu'elles soient clairement

spécifiées. Enfin, le code écrit devra être sûr (pas d'accès invalide à un tableau, pas de division par zéro, et le programme termine, notamment) pour toutes valeurs des paramètres vérifiant les conditions données dans l'énoncé.

Le temps de calcul d'une procédure `proc` de paramètres $p_1, \ldots, p_k$ est défini comme le nombre d'opérations (accès en lecture ou écriture à une case d'un tableau ou à une variable, appel à une des primitives données dans l'énoncé) exécutées par `proc` pour ces paramètres ; on note $T(\texttt{proc}, n)$ le temps de calcul maximal pris sur tous les paramètres possibles pour n fixé. On dit que `proc` s'exécute en temps linéaire si il existe des réels $\alpha, \beta > 0$ et un entier $N \geq 0$ tels que $\alpha.n \leq T(\texttt{proc}, n) \leq \beta.n$ pour tout $n \geq N$. De même, on dit que `proc` s'exécute en temps logarithmique si il existe des réels $\alpha, \beta > 0$ et un entier $N \geq 0$ tels que $\alpha \log n \leq T(\texttt{proc}, n) \leq \beta \log n$ pour tout $n \geq N$.

Partie I. Recherche de point fixe : cas général

On rappelle que x est un *point fixe* de la fonction f si et seulement si $f(x) = x$.

Question 1 Écrire une procédure `admet_point_fixe(t)` qui prend en argument un tableau `t` de taille n et renvoie `vrai` si la fonction $f\colon \mathsf{E}_n \to \mathsf{E}_n$ représentée par `t` admet un point fixe, `faux` sinon. Par exemple, `admet_point_fixe` devra renvoyer `vrai` pour le tableau donné en introduction, puisque 9 est un point fixe de la fonction f_0 qui à x associe $2x + 1$ modulo 10.

Question 2 Écrire une procédure `nb_points_fixes(t)` qui prend en argument un tableau `t` de taille n et renvoie le nombre de points fixes de la fonction $f\colon \mathsf{E}_n \to \mathsf{E}_n$ représentée par `t`. Par exemple, `nb_points_fixes` devra renvoyer 1 pour le tableau donné en introduction, puisque 9 est le seul point fixe de f_0.

On note f^k l'itérée k-ième de f, autrement dit

$$\begin{array}{rccl} f^k\colon & \mathsf{E}_n & \to & \mathsf{E}_n \\ & x & \mapsto & \underbrace{f(f(\ldots f}_{k \text{ fois}}(x))\ldots) \ . \end{array}$$

Question 3 Écrire une procédure `itere(t,x,k)` qui prend en premier argument un tableau `t` de taille n représentant une fonction $f\colon \mathsf{E}_n \to \mathsf{E}_n$, en deuxième et troisième arguments des entiers x, k de E_n, et renvoie $f^k(x)$.

Question 4 Écrire une procédure `nb_points_fixes_iteres(t,k)` qui prend en premier argument un tableau `t` de taille n représentant une fonction $f\colon \mathsf{E}_n \to \mathsf{E}_n$, en deuxième argument un entier $k \geq 0$, et renvoie le nombre de points fixes de f^k.

Un élément $z \in \mathsf{E}_n$ est dit *attracteur principal* de $f : \mathsf{E}_n \to \mathsf{E}_n$ si et seulement si z est un point fixe de f, et pour tout $x \in \mathsf{E}_n$, il existe un entier $k \geq 0$ tel que $f^k(x) = z$.

Afin d'illustrer cette notion, on pourra vérifier que la fonction f_1 représentée par le tableau

ci-dessous admet 2 comme attracteur principal.

5	5	2	2	0	2	2
t[0]	t[1]	t[2]	t[3]	t[4]	t[5]	t[6]

En revanche, on notera que la fonction f_0 donnée en introduction n'admet pas d'attracteur principal, puisque $f_0^k(0) \neq 9$ quel que soit l'entier $k \geq 0$.

Question 5 Écrire une procédure `admet_attracteur_principal(t)` qui prend en argument un tableau `t` de taille n et renvoie `vrai` si et seulement si la fonction $f\colon \mathsf{E}_n \to \mathsf{E}_n$ représentée par `t` admet un attracteur principal, `faux` sinon. On ne requiert pas ici une solution efficace.

On suppose aux questions 6 et 7 que f admet un attracteur principal. Le *temps de convergence* de f en $x \in \mathsf{E}_n$ est le plus petit entier $k \geq 0$ tel que $f^k(x)$ soit un point fixe de f. Pour la fonction f_1 ci-dessus, le temps de convergence en 4 est égal à 3. En effet, $f_1(4) = 0$, $f_1^2(4) = 5$, $f_1^3(4) = 2$, et 2 est un point fixe de f_1. On note $\mathsf{tc}(f, x)$ le temps de convergence de f en x.

Question 6 Écrire une procédure `temps_de_convergence(t,x)` qui prend en premier argument un tableau `t` de taille n représentant une fonction $f\colon \mathsf{E}_n \to \mathsf{E}_n$ qui admet un attracteur principal, en deuxième argument un entier x de E_n, et renvoie le temps de convergence de f en x. On pourra admettre que $\mathsf{tc}(f, x)$ vaut 0 si x est un point fixe de f, et $1 + \mathsf{tc}(f, f(x))$ si x n'est pas un point fixe de f.

Question 7 Écrire une procédure `temps_de_convergence_max(t)` qui prend en argument un tableau `t` de taille n représentant une fonction $f\colon \mathsf{E}_n \to \mathsf{E}_n$ qui admet un attracteur principal, et renvoie $\max_{i=0\ldots n-1} \mathsf{tc}(f, i)$. On impose un temps de calcul <u>linéaire</u> en la taille n du tableau. À titre d'indication, on pourra au besoin créer un deuxième tableau, qui servira d'intermédiaire au cours du calcul. On ne demande pas de démonstration du fait que le temps de calcul de la solution proposée est linéaire.

Partie II. Recherche efficace de points fixes

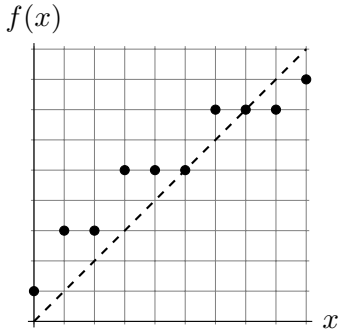
Toute procédure `point_fixe(t)` retournant un point fixe d'une fonction arbitraire est de complexité au mieux linéaire en n. On s'intéresse maintenant à des améliorations possibles de cette complexité lorsque la fonction considérée possède certaines propriétés spécifiques. Nous examinons deux cas.

Premier cas.

Le premier cas que nous considérons est celui d'une fonction croissante de E_n dans E_n. On rappelle qu'une fonction $f\colon \mathsf{E}_n \to \mathsf{E}_n$ est croissante si et seulement si pour tous $x, y \in \mathsf{E}_n$ tels que $x \leq y$, $f(x) \leq f(y)$.

On admet qu'une fonction croissante de E_n dans E_n admet toujours un point fixe.

À titre d'exemple, la fonction dont le tableau et le graphe sont donnés ci-dessous est croissante. Elle a deux points fixes, à savoir les entiers 5 et 7.



1	3	3	5	5	5	7	7	7	8
t[0]	t[1]	t[2]	t[3]	t[4]	t[5]	t[6]	t[7]	t[8]	t[9]

Question 8 Écrire une procédure `est_croissante(t)` qui prend en argument un tableau `t` de taille n et renvoie `vrai` si la fonction représentée par `t` est croissante, `faux` sinon. On impose un temps de calcul linéaire en la taille n du tableau. On ne demande pas de démonstration du fait que le temps de calcul de la solution proposée est linéaire.

Question 9 Écrire une procédure `point_fixe(t)` qui prend en argument un tableau `t` de taille n représentant une fonction croissante $f\colon \mathsf{E}_n \to \mathsf{E}_n$, et retourne un entier $x \in \mathsf{E}_n$ tel que $f(x) = x$. On impose un temps de calcul logarithmique en la taille n du tableau. On ne demande pas ici de démonstration du fait que le temps de calcul de la solution proposée est logarithmique, ceci étant le sujet de la question suivante.

Question 10 Démontrer que la procédure de la question 9 termine. On rappelle que pour prouver qu'une boucle termine, il suffit d'exhiber un entier positif i, fonction des variables du programme, qui décroît strictement à chaque itération de boucle. Justifier que le temps de calcul est logarithmique en la taille n du tableau.

Deuxième cas.

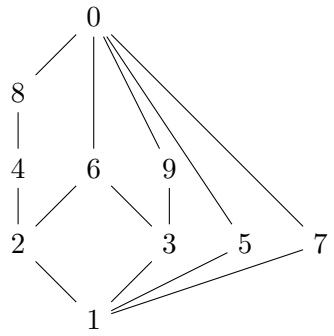
On peut généraliser la notion de fonction croissante comme suit. On rappelle qu'une relation binaire $\preceq$ sur un ensemble E est une relation d'ordre si et seulement si elle est réflexive ($x \preceq x$ pour tout $x \in E$), anti-symétrique (pour tous $x, y \in E$, si $x \preceq y$ et $y \preceq x$, alors $x = y$), et transitive (pour tous $x, y, z \in E$, si $x \preceq y$ et $y \preceq z$, alors $x \preceq z$). Soit $\preceq$ une relation d'ordre sur E. Une fonction $f\colon E \to E$ est *croissante au sens de* $\preceq$ si et seulement si pour tous $x, y \in E$, $x \preceq y$ implique $f(x) \preceq f(y)$.

Ceci généralise la notion de fonction croissante de E_n dans E_n, que l'on retrouve en prenant $E = \mathsf{E}_n$ et $\preceq$ la relation d'ordre $\leq$. On s'intéresse dorénavant à d'autres relations d'ordre sur E_n.

On dit qu'un élément m de E est un *plus petit élément* de E au sens de $\preceq$ si et seulement si, pour tout $x \in E$, $m \preceq x$. On admet que pour tout ensemble fini E, muni d'une relation d'ordre $\preceq$ et qui admet un plus petit élément m au sens de $\preceq$, pour toute fonction croissante $f\colon E \to E$ au sens de $\preceq$, il existe un entier $k \geq 0$ tel que $f^k(m)$ est un point fixe de f dans E.

Question 11 Soit E un ensemble fini quelconque muni d'une relation d'ordre $\preceq$ et admettant un plus petit élément m au sens de $\preceq$. Soit $f\colon E \to E$ une fonction croissante au sens de $\preceq$, et soit $k \geq 0$ un entier tel que $f^k(m)$ soit un point fixe de f dans E. Démontrer que $f^k(m)$ est en fait le plus petit point fixe de f au sens de $\preceq$, autrement dit que pour tout autre point fixe x de f dans E, on a $f^k(m) \preceq x$.

Nous nous intéressons maintenant à un choix particulier d'ordre $\preceq$, appelé *ordre de divisibilité* et noté $|$. Précisément, on note $a \mid b$ la relation d'ordre "a divise b" sur les entiers positifs, vraie si et seulement s'il existe un entier $c \geq 0$ tel que $ca = b$. Ainsi, l'ensemble E_{10} ordonné par divisibilité peut se représenter graphiquement comme suit.



D'après la définition donnée précédemment, une fonction $f\colon \mathsf{E}_n \to \mathsf{E}_n$ *croissante au sens de l'ordre de divisibilité* est une fonction telle que pour tous x, y dans E_n, si $x \mid y$, alors $f(x) \mid f(y)$. Par exemple, la fonction représentée par le tableau ci-dessous est croissante au sens de l'ordre de divisibilité.

0	2	4	6	4	8	0	2	0	6
`t[0]`	`t[1]`	`t[2]`	`t[3]`	`t[4]`	`t[5]`	`t[6]`	`t[7]`	`t[8]`	`t[9]`

.

On remarque que, par la question 11, toute fonction de E_n dans E_n croissante au sens de l'ordre de divisibilité a un plus petit point fixe au sens de l'ordre de divisibilité.

On rappelle que le pgcd de deux entiers $x \geq 1$ et $y \geq 1$ est le plus grand entier non nul qui divise x et y. On étend cette définition à des entiers naturels quelconques, en convenant de définir le pgcd d'un entier $x \geq 0$ et de 0 comme valant x.

Question 12 Soit f une fonction de E_n dans E_n, croissante au sens de l'ordre de divisibilité, et notons $x_1, \ldots, x_m$ les points fixes de f dans E_n. Montrer que le plus petit point fixe de f au sens de l'ordre de divisibilité est exactement le pgcd de $x_1, \ldots, x_m$.

Question 13 Écrire une procédure `pgcd_points_fixes(t)` qui prend en argument un tableau `t` de taille n représentant une fonction de E_n dans E_n, croissante au sens de la divisibilité, et renvoie le pgcd de ses points fixes. On impose un temps de calcul logarithmique en la taille n du tableau. On ne demande pas ici de démonstration du fait que le temps de calcul de la solution proposée est logarithmique, ceci étant le sujet de la question qui suit.

Question 14 Justifier que la procédure de la question 13 a un temps de calcul logarithmique en la taille n du tableau.

* *
*

3 X/Cachan 2013

« Primitives »

En informatique, on désigne par primitive d'un langage un objet qui n'est pas construit à partir du langage lui-même. Par exemple en OCaml, il existe des primitives écrites en C.
Ici, nous allons construire les fonctions évoquées dans le préambule du sujet en Maple donc ce n'est pas vraiment des primitives mais cela nous permettra de nous « remettre dans le bain » onctueux de MAPLE...

allouer

La fonction `allouer(m,c)` renvoie un tableau de taille m rempli de c. Comme il s'agit de tableaux d'une ligne, nous allons plutôt nous tourner sur des objets de type *liste*, facilement manipulables en MAPLE, mais il faut bien faire la différence entre les listes qui sont définies dynamiquement et les tableaux qui sont définis statiquement. Le jour du concours, le jury accepte que vous employiez des listes au lieu de tableaux `array` si vous le précisez et le motivez.

Maple
```
> allouer := proc(m::nonnegint,c)
    RETURN([c$m])
  end:
```

Il est optionnel de préciser le type des arguments des fonctions mais cela sert de garde-fou. Notez l'utilisation du `$`

taille

Il existe une fonction qui compte le nombre d'opérandes d'une liste : `nops`.

Exemple

Nous utiliserons le tableau du sujet comme exemple de référence pour nos tests. Nous n'aurons donc pas besoin de la fonction `allouer`...

Maple
```
> tab_ex := [1,3,5,7,9,1,3,5,7,9]:
```

Danger

Maple commence à compter à 1 alors que le sujet précise que le premier indice des tableaux est 0 : vous pouvez donc **à l'écrit** considérer que Maple commence à compter à 0 mais sur vos tests sur machine aujourd'hui, ce **`@@@!!###!!`** Maple voudra que vous commenciez à 1....

Passage par valeur / par adresse

Au début du sujet, il est évoqué le passage de paramètre, ce qui est grandement hors programme et donc déroutant !
En gros, il existe deux types de passage des paramètres d'une fonction :

par valeur (ou par copie) : la fonction ne dispose que d'une copie de son paramètre. C'est cette copie qu'elle peut modifier mais le paramètre d'entrée peut rester, lui, immuable (non « mutable » en franglais informatique).

par adresse : la fonction dispose de l'adresse de son argument. Elle peut donc modifier le contenu de cette adresse et à la fin de l'exécution l'argument se trouve irrémédiablement modifié. C'est plus rapide mais dangereux car cela crée un *effet de bord*. Le paramètre d'entrée ne peut plus être immuable (il doit être « mutable »).

Voyons ce que cela donne en Maple avec une liste (immuable) ou un tableau (mutable). On prend un paramètre qui sera un tableau ou une liste d'entiers et on modifie son premier terme :

Maple

```
> test := proc(t)
    t[1]:= t[1]+2;
    RETURN(t);
  end:
```

Avec un tableau :

Maple

```
> tab := array([3,4,5,6]):
> test(tab):
> tab[1]
                                5
> test(tab):
> tab[1]
                                7
```

Le paramètre a été irrémédiablement modifié par la fonction. De plus, `tab[1]` ne renvoie pas toujours la même chose !...
Avec une liste :

Maple

```
> liste := [3,4,5,6]:
> test(liste):
Error, (in test) Illegal use of a formal parameter
```

Il faut donc faire le passage par valeur à la main.

Maple

```
> test2 := proc(t)
    local u;
    u := t;
    u[1]:= t[1]+2;
    RETURN(u);
  end:
```

Alors :

Maple

```
> liste := [3,4,5,6]:
> test2(liste);
                            [5, 4, 5, 6]
> liste;
                            [3, 4, 5, 6]
> test2(liste);
                            [5, 4, 5, 6]
```

OK, mais est-ce que ça change quelque chose pour les tableaux :

Maple

```
> test2(tab):
> tab[1]
                                5
> test2(tab):
```

```
> tab[1]
                                7
```

Maple continue donc à passer les arguments par adresse même si nous avons créé une variable locale pour faire à la main un passage par valeur.

Bref, encore une fois, programmer avec Maple qui n'est pas un langage de programmation, c'est sportif...

Pour ne pas se perdre, je vous conseille d'utiliser des listes et de créer à la main des variables locales.

Question 1

On pensera à utiliser **RETURN** qui permet de sortir de la procédure sans aller au bout de la boucle.

Pour se fixer notre façon de travailler, voici ce que nous écrirons sur Maple :

Maple

```
> admet_point_fixe := proc(t::list)
    local k;
    for k from 1 to nops(t) do
          if t[k] = k - 1
          then RETURN(true)
          fi;
   od;
 RETURN(false);
 end:
```

et sur la copie :

Maple

```
> admet_point_fixe := proc(t::list)
    local k;
    for k from 1 to taille(t) do
          # est-ce que k est un point fixe ?
          if t[k] = k
          # on renvoie vrai et on arrête l'exécution
          then RETURN(vrai)
          fi;
   od;
 # si l'on n'est pas déjà sorti, c'est qu'il n'y a pas de point fixe
 RETURN(faux);
 end:
```

Dans de vrais langages de programmation, cela peut être plus clair et concis. Par exemple en Haskell :

Haskell
```
> any (\x -> mod (2*x + 1) 10 == x) [1..9]
True
```

Aparté

Mais bon, cela réduirait le contenu de ce sujet...En prépa, on apprend aux futurs ingénieur(e)s à programmer de manière inutilement compliquée avec des outils préhistoriques ;-)
Mais je suis un peu de mauvaise foi car Maple intègre deux procédures fort utiles héritées des principes de la programmation fonctionnelle : **map** et **select**.
Par exemple, ici :

Maple
```
> select(x -> (2*x+1) mod 10 = x, [k $ k = 1..9]);
                                [9]
```

Question 2

On reprend la fonction précédente mais en introduisant un compteur.

Question 3

C'est l'occasion de voir un peu de récursivité (une définition avec des ... est une définition récursive : c'est donc l'énoncé qui nous y incite...).
Par exemple, $f^5(1) = f^4(f(1))$ i.e. **itere(t,1,5) = itere(t,3,4)**.
Attention aux indices...

Question 4

Il n'y a pas grand chose à faire si ce n'est du copier-coller....

Question 5

Deux petites questions préliminaires : combien de points fixes peut admettre une fonction ayant un attracteur principal ? En combien d'itérations au maximum cet attracteur est-il atteint à partir d'un entier de E_n ?
N'oubliez pas non plus de lire la dernière phrase de la question...

Question 6

L'énoncé donne une relation de récurrence...Décidément, difficile d'échapper à la récursion dans ce sujet, n'en déplaise à certains correcteurs...
Par sécurité, on pourrait introduire un message d'erreur s'il n'y a pas d'attracteur principal avec **ERROR(`Pas d'attracteur principal`)**.

Question 7

On répond à ce genre de question en une ligne de manière claire et efficace avec Haskell ou OCaml mais bon...
La question est bizarrement posée car la fonction précédente est déjà au pire de complexité linéaire et on ne voit pas comment ne pas calculer chaque temps de convergence donc cela nous donne une complexité quadratique...
Bon, on va supposer que le rédacteur pense à la complexité de cette fonction sans tenir compte de celle des autres.
Maple sait comparer deux nombres avec **max**.

Question 8

C'est un peu le même principe que la fonction précédente.

Question 9

Alors là, c'est carrément hors programme cette histoire de complexité logarithmique, à croire que les rédacteurs des sujets pour étudiants de MP-SI pensent qu'ils écrivent un sujet pour MP-INFO.
Des étudiants d'option INFO, quand ils voient « complexité logarithmique », pensent d'abord à « dichotomie » puis ensuite éventuellement cherchent ailleurs...

Question 10

Rigolo le rappel sur du hors programme...En grec, $\delta\iota\chi\sigma\tau\sigma\mu\iota\alpha$ signifie « couper en deux » : qu'est-ce qui est coupé en deux dans `point_fixe` ?

Question 11

Que pensez-vous de $f^k(x)$?

Fin

Des questions d'arithmétique...

4 X/Cachan 2010

Voici encore un sujet qui est allé hors programme en parlant d'algorithmes de tri.
Nous allons encore travailler sur des listes plutôt qu'avec des tableaux. On a souvent besoin de considérer la *tête* d'une liste qui est son premier élément et la *queue* d'une liste qui est la liste privée de son premier élément.
Nous pouvons donc introduire deux primitives :

Maple

```
> Tete := proc(L)
   RETURN(L[1])
  end:
> Queue := proc(L)
   RETURN(L[2..-1])
  end:
```

Pour la beauté du geste, vous essaierez d'éviter les boucles *pour* ou *tant que* pour répondre aux 5 premières questions au moins.

1. Il est plus léger ici d'utiliser l'algorithme de Hörner pour évaluer un polynôme. 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 :

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

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).

Proposez ici une version récursive utilisant la liste des coefficients et nos primitives `Tete` et `Queue`.

2. Encore une fois, une version récursive est rapide à écrire. Vous aurez peut-être besoin de distinguer trois conditions. Utilisez alors la structure conditionnelle :

Maple
```
if ... then ...
elif ... then ...
else ...
fi
```

3. Une version récursive est attendue. Vous aurez sûrement besoin de concaténer deux listes ou d'ajouter un élément à une liste.

   Pour ajouter un 0 en bout de liste par exemple, on peut faire :

Maple
```
liste := [op(liste),0]
```

4. Une structure conditionnelle `if...elif...else...fi` traduit l'énoncé. On pourra utiliser la commande `signum(n)` qui renvoie 1 si $n > 0$, -1 si $n < 0$ et 0 sinon.

5. Voici une petite question qui cache un chapitre de l'option informatique consacrée aux tris !... Le sujet est vaste... Pour votre culture, nous aborderons tout de même deux tris possibles et leur complexité : le tri par insertion et le tri fusion.

   Tri par insertion C'est le tri qui correspond à un joueur qui classe ses cartes.

   Nous privilégierons une version récursive bien sûr.

   Commencez par créer une procédure récursive qui insère un élément dans une liste. Comme vous avez lu tout le sujet, vous savez qu'à la question suivante, il sera utile de trier une liste avec deux relations d'ordre différentes.

   Vous allez donc créer une procédure `insere := proc(Ordre,P,Liste)` qui insère le polynôme `P` dans la liste de polynômes `Liste` selon l'ordre croissant correspondant à `Ordre` (qui sera par exemple `Compare_neg` dans cette question).

   Pour obtenir la liste triée, il suffit de trier un à un les éléments de la liste avec une procédure récursive `tri := proc(Ordre,Liste)` qui utilise `insere`.

   Par exemple, vous devriez obtenir :

Maple
```
> tri(Compare_neg,[[1],[0,-1],[0,1]]);
                        [[0, 1], [0, -1], [1]]
```

   qui correspond à la sixième figure proposée dans le sujet.

   Combien d'opérations élémentaires la procédure va-t-elle effectuer dans le pire des cas ?

   Tri fusion On utilise la méthode diviser pour régner : on « coupe » le tableau à trier en deux, on trie chaque tableau et on fusionne les deux tableaux obtenus, de manière récursive.

   Il faut donc commencer par créer une procédure

   `divise := proc(Ordre,L)`

qui divise la liste `L` en deux listes de longueurs égales ou presque, selon la parité de la taille.

Il faudra ensuite créer une procédure récursive

`fusionne := proc(Ordre,L1,L2)`

qui prend deux listes triées et en fait une seule liste triée.

On utilise ensuite ces deux procédures pour créer une procédure récursive

`tri_f := proc(Ordre,Liste)`

qui trie la liste `L`.

Pour la *complexité*, on peut supposer que la division s'effectue à temps constant et que la fusion de deux listes triées est de l'ordre de la taille n du tableau.

Soit K_n la complexité du tri d'une liste de taille n.

On obtient donc que K_n est de l'ordre de $2K_{\lfloor n/2 \rfloor} + n$.

Il s'agit donc d'étudier la suite de terme général $u_n = 2u_{\lfloor n/2 \rfloor} + n$ avec $u_0 = u_1 = 0$.

Montrez que u_n est croissante.

En introduisant la suite définie par $x_k = u_{2^k}$, montrez que :

$$n\lfloor \log_2(n) \rfloor \leqslant u_n \leqslant 2n\left(\lfloor \log_2(n) \rfloor + 1\right)$$

6. Tout est expliqué... Après ce gros effort de récursion, vous pouvez utiliser une boucle *pour*. Utilisez à bon escient `RETURN` qui permet de sortir de la procédure, ce qui est adapté à ces procédure de vérification pour économiser du temps : on en sort dès qu'on tombe sur un os.

Maple
```
> verifier_permute([2,3,1],[[0,1],[0,-1],[1]]);
                                true
```

7. On peut utiliser ici une triple boucle *pour* indexée sur les coefficients a, b et c.
   Attention ! $<$ est une relation d'ordre binaire en Maple ce qui interdit l'utilisation directe de `a<b<c<d`. On passera par :

   `(x<y)and (y<z)`

   Pour traduire la double inégalité $x < y < z$.

Maple
```
> est_echangeur_aux([2,4,1,3],2);
                                true
> est_echangeur_aux([2,4,1,3],1);
                               false
```

8. Pas de difficulté particulière :

Maple
```
> est_echangeur([2,4,1,3]);
                               false
```

9. L'utilisation de `sum` dans une récursion est peu fructueuse en Maple. On préférera une double boucle.

Maple
```
> nombre_echangeurs(4);
                                 22
```

10. On traduit simplement l'énoncé.
11. C'est la question compliquée...

Maple

```
> enumerer_echangeurs(4);
[
[1,2,3,4],[1,2,4,3],[1,3,2,4],[1,3,4,2],[1,4,2,3],[1,4,3,2],[2,1,3,4],
[2,1,4,3],[2,3,1,4],[2,3,4,1],[2,4,3,1],[3,1,2,4],[3,2,1,4],[3,2,4,1],
[3,4,1,2],[3,4,2,1],[4,1,2,3],[4,1,3,2],[4,2,1,3],[4,2,3,1],[4,3,1,2],
[4,3,2,1]
]
```

ÉCOLE POLYTECHNIQUE

ÉCOLE SUPÉRIEURE DE PHYSIQUE ET CHIMIE INDUSTRIELLES

CONCOURS D'ADMISSION 2010

FILIÈRE MP - OPTION PHYSIQUE ET SCIENCES DE L'INGÉNIEUR **FILIÈRE PC**

COMPOSITION D'INFORMATIQUE

(Durée : 2 heures)

L'utilisation des calculatrices **n'est pas autorisée** pour cette épreuve.
Le langage de programmation choisi par le candidat doit être spécifié en tête de la copie.

⋆ ⋆ ⋆

Échangeurs de Polynômes

Dans ce problème on s'intéresse à des polynômes à coefficients réels qui s'annulent en 0. Un tel polynôme P s'écrit donc $P(x) = a_1x + a_2x^2 + \cdots + a_mx^m$. Le but de ce problème est d'étudier la position relative autour de l'origine de plusieurs polynômes de ce type.

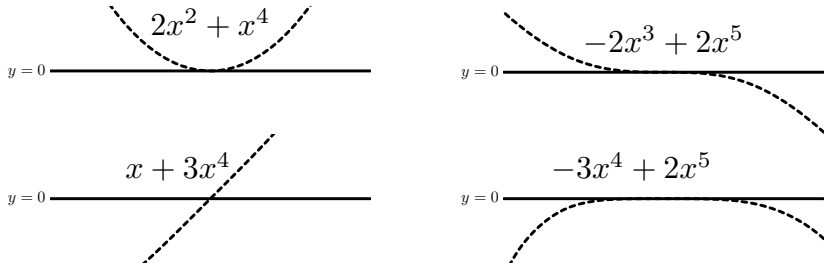
Dans tout le problème, les polynômes sont représentés par des tableaux de nombres flottants de la forme `[`a_1`;`a_2`;`$\cdots$`;`a_m`]`. Le nombre a_m peut être nul, par conséquent un polynôme donné admet plusieurs représentations sous forme de tableau. Ces tableaux sont indexés à partir de 1 et les éléments d'un tableau de taille m sont donc indexés de 1 à m. On suppose qu'il existe également une primitive `allouer(`m`)` pour créer un tableau de m cases. La taille m d'un tableau t est renvoyée par la primitive `taille(`t`)`. L'accès à la $i^{\text{ème}}$ case d'un tableau t est noté $t[i]$. Par ailleurs, on suppose que les tableaux peuvent être passés en argument ou renvoyés comme résultat de fonction, quel que soit le langage utilisé par le candidat pour composer. Enfin, les booléens `vrai` et `faux` sont utilisés dans certaines questions de ce problème. Le candidat est libre d'utiliser les notations propres à ces booléens dans le langage dans lequel il compose.

Le problème est découpé en deux parties qui peuvent être traitées de manière indépendante. Cependant, la **partie II** utilise les notions et notations introduites dans la **partie I**.

I. Permutation de n polynômes

Question 1 Afin de se familiariser avec cette représentation, écrire une fonction `evaluation` qui prend en arguments un polynôme P, représenté par un tableau, et un nombre flottant v, et qui renvoie la valeur de $P(v)$.

Nous commençons notre étude par quelques observations. Voici des exemples de graphes de polynômes autour de l'axe des abscisses.

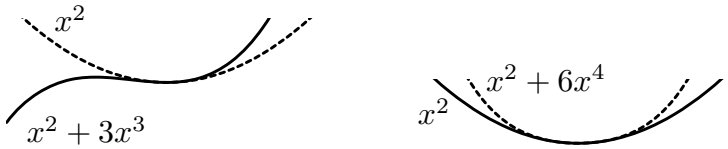


On remarque que le comportement au voisinage de l'origine est décrit par le premier monôme $a_k x^k$ dont le coefficient a_k est non nul (les coefficients a_1, ..., a_{k-1} étant donc tous nuls). En effet, quand x est petit, le terme $a_{k+1}x^{k+1} + \ldots + a_m x^m$ est négligeable devant le terme $a_k x^k$. Cet entier k est la *valuation* du polynôme à l'origine. Par exemple, la valuation du polynôme $-2x^3 - 3x^5 + 4x^7$ est 3. On remarque alors les deux règles suivantes au voisinage de l'origine :

- Si la valuation k est *paire*, le graphe du polynôme reste du *même* côté de l'axe des abscisses.
- Si la valuation k est *impaire*, le graphe du polynôme *traverse* l'axe des abscisses.

Question 2 Écrire une fonction `valuation` qui prend en argument un polynôme P et renvoie sa valuation. Par définition, cette fonction renverra 0 si P est le polynôme nul.

On s'intéresse maintenant aux positions relatives autour de l'origine des graphes de deux polynômes P_1 et P_2. La figure suivante montre les graphes de polynômes autour de l'origine.



On remarque que le comportement de ces graphes dépend de la parité de la valuation de la différence $P_1 - P_2$:

- Si la valuation de $P_1 - P_2$ est *paire*, les deux graphes se touchent mais ne se traversent pas à l'origine.
- Si la valuation de $P_1 - P_2$ est *impaire*, les deux graphes se traversent à l'origine.

Question 3 Écrire une fonction `difference` qui prend en arguments deux polynômes P_1 et P_2 (dont les tailles peuvent être différentes) et qui renvoie la différence des polynômes $P_1 - P_2$.

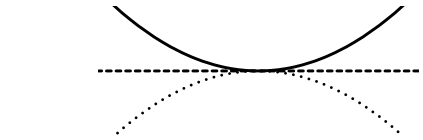
Question 4 Écrire une fonction `compare_neg` qui prend en arguments deux polynômes P_1 et P_2 et qui renvoie :

- un entier strictement négatif si $P_1(x)$ est plus petit que $P_2(x)$, pour x *négatif* assez petit
- 0 si les deux polynômes P_1 et P_2 sont égaux
- un entier strictement positif si $P_1(x)$ est plus grand que $P_2(x)$, pour x *négatif* assez petit.

On admettra sans démonstration que la fonction `compare_neg` définit une relation d'ordre.

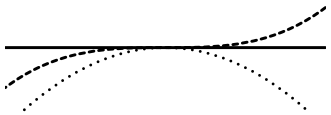
Enfin, passons à l'étude des graphes de trois polynômes. Les figures ci-après montrent les positions relatives de trois polynômes P_1, P_2 et P_3 autour de l'origine, avec la légende suivante :

$P_1(x)$ —— $P_2(x)$ ------- $P_3(x)$ ·····

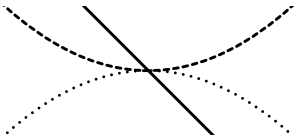


$P_1(x) = x^2 \quad P_2(x) = 0 \quad P_3(x) = -x^2$

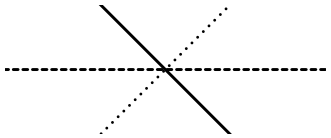
$P_1(x) = x^2 \quad P_2(x) = -x^3 \quad P_3(x) = 0$



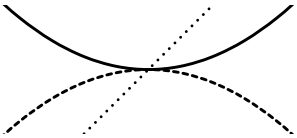
$P_1(x) = 0 \quad P_2(x) = x^3 \quad P_3(x) = -x^2$



$P_1(x) = -x \quad P_2(x) = x^2 \quad P_3(x) = -x^2$



$P_1(x) = -x \quad P_2(x) = 0 \quad P_3(x) = x$



$P_1(x) = x^2 \quad P_2(x) = -x^2 \quad P_3(x) = x$

Le choix de ces polynômes est fait pour qu'à chaque fois les inégalités $P_1(x) > P_2(x) > P_3(x)$ soient vérifiées pour x légèrement négatif. Maintenant, observons les positions relatives de ces graphes pour x légèrement positif. On remarque que l'ordre des courbes est *permuté* : on passe de l'ordre $P_1(x) > P_2(x) > P_3(x)$ à un autre ordre. La donnée des trois polynômes P_1, P_2 et P_3 définit donc une unique *permutation* π de $\{1, 2, 3\}$ telle que $P_{\pi(1)}(x) > P_{\pi(2)}(x) > P_{\pi(3)}(x)$, pour x positif et assez petit. On note que les *six* permutations de $\{1, 2, 3\}$ sont possibles, comme le montrent les six exemples ci-dessus.

De manière générale, on dit qu'une permutation π de $\{1, 2, \ldots, n\}$ *permute* les polynômes P_1, P_2, …, P_n si et seulement si :

$$\begin{aligned} & P_1(x) > P_2(x) > \ldots > P_n(x) && \text{pour } x \textit{ négatif} \text{ assez petit} \\ \text{et} \quad & P_{\pi(1)}(x) > P_{\pi(2)}(x) > \ldots > P_{\pi(n)}(x) && \text{pour } x \textit{ positif} \text{ assez petit} \end{aligned}$$

Ce qui était vrai pour trois polynômes ne l'est plus à partir de quatre polynômes : il existe des permutations qui ne permutent aucun ensemble de polynômes P_1, P_2, …, P_n.

Dans la suite, les permutations de $\{1, 2, \ldots, n\}$ seront représentées par des tableaux d'entiers de taille n, indexés à partir de 1 et contenant tous les entiers entre 1 et n.

Question 5 Écrire une fonction `tri` qui prend en argument un tableau t contenant n polynômes et qui le trie en utilisant la fonction `compare_neg`, de telle sorte que l'on ait $t[1](x) > t[2](x) > \cdots > t[n](x)$ pour x négatif et assez petit. Le candidat ne pourra pas utiliser pour cette question de fonction de tri prédéfinie dans la bibliothèque du langage qu'il utilise pour composer.

Question 6 Écrire une fonction `verifier_permute` qui prend en argument une permutation π de $\{1, 2, \ldots, n\}$ et un tableau t de même taille supposé trié par la fonction `tri`, et renvoie `vrai` si π permute les n polynômes $t[1]$, $t[2]$, …, $t[n]$ contenus dans t, et `faux` sinon. On pourra s'aider d'une fonction `compare_pos`, similaire à la fonction `compare_neg`, pour comparer deux polynômes pour x *positif* assez petit.

II. Échangeurs de n polynômes

Dans la suite, nous dirons qu'une permutation π de $\{1, 2, \ldots, n\}$ est un *échangeur* s'il existe n polynômes P_1, P_2, ..., P_n tels que π permute ces polynômes. Nous allons maintenant écrire des fonctions qui répondent aux questions suivantes : Une permutation π est-elle un échangeur ? Peut-on dénombrer les échangeurs ? Peut-on énumérer les échangeurs ?

Une condition nécessaire et suffisante pour qu'une permutation soit un échangeur est la suivante : une permutation π de $\{1, 2, \ldots, n\}$ est un échangeur si et seulement si il n'existe aucun entiers a, b, c, d tels que $n \geq a > b > c > d \geq 1$ et

$$\pi(b) > \pi(d) > \pi(a) > \pi(c) \quad \text{ou} \quad \pi(c) > \pi(a) > \pi(d) > \pi(b) \tag{1}$$

Question 7 Écrire une fonction `est_echangeur_aux` qui prend en argument une permutation π de $\{1, 2, \ldots n\}$ et un entier d tel que $1 \leq d \leq n$ et qui renvoie `vrai` s'il n'existe aucun entier a, b et c tels que $n \geq a > b > c > d$ et vérifiant (1), et `faux` sinon.

Question 8 En utilisant la fonction `est_echangeur_aux`, écrire une fonction `est_echangeur` qui prend en argument une permutation π de $\{1, 2, \ldots, n\}$ et renvoie `vrai` si π est un échangeur, et `faux` sinon.

On admet sans démonstration que la relation de récurrence suivante permet de compter le nombre $a(n)$ de permutations de $\{1, 2, \ldots, n\}$ qui sont des échangeurs :

$$a(1) = 1, \qquad a(n) = a(n-1) + \sum_{i=1}^{n-1} a(i) \times a(n-i)$$

Question 9 Écrire une fonction `nombre_echangeurs` qui prend un entier n en argument et renvoie le nombre d'échangeurs $a(n)$. Enfin, les deux questions suivantes ont pour but d'énumérer tous les échangeurs de $\{1, 2, \ldots, n\}$.

Question 10 Écrire une fonction `decaler` qui prend en arguments un tableau t de taille n et un entier v, et renvoie un nouveau tableau u de taille $n + 1$ tel que

$$\begin{cases} u[1] = v \\ u[i] = t[i-1] & \text{si } t[i-1] < v \text{ et } 2 \leq i \leq n+1 \\ u[i] = 1 + t[i-1] & \text{si } t[i-1] \geq v \text{ et } 2 \leq i \leq n+1 \end{cases}$$

L'algorithme que nous allons utiliser pour énumérer les échangeurs de $\{1, 2, \ldots, n\}$ consiste à énumérer successivement les échangeurs de $\{1, 2, \ldots, k\}$, pour tout k de 1 à n, dans un tableau t de taille $a(n)$. Si on suppose qu'un tableau t contient les m échangeurs de $\{1, \ldots, k\}$ entre les cases $t[1]$ et $t[m]$, on peut en déduire les échangeurs de $\{1, \ldots, k+1\}$ de la manière suivante : pour tout entier v entre 1 et $k+1$ et tout entier i entre 1 et m, on décale (à l'aide de la fonction `decaler`) l'échangeur $t[i]$ avec v puis on teste si le résultat est un échangeur (avec la fonction `est_echangeur_aux`).

Question 11 Écrire une fonction `enumerer_echangeurs` qui prend un entier n en argument et renvoie un tableau contenant les $a(n)$ échangeurs de $\{1, 2, \ldots, \mathtt{n}\}$. On pourra utiliser un second tableau pour stocker temporairement les nouveaux échangeurs.

* *
*

TP N°

2

L'aventure géométrique

À la fin des années 1960 au MIT, Seymour PAPERT, après avoir travaillé avec Jean PIAGET, met au point un langage de programmation, LOGO, pour mettre en pratique la thèse qu'il défendra pendant des années : « c'est en créant qu'on apprend ». L'informatique devient alors un outil d'**exploration** de la mathématique. C'est dans cet état d'esprit que nous allons expérimenter des concepts mathématiques très importants : les rapports global/local, implicite/explicite, topologie/-géométrie,...et tout ça avec une petite tortue que nous allons piloter depuis Maple pour aller effectuer un voyage fantastique au centre de la mathématique.

Pour prolonger vos explorations, n'hésitez pas à dévorer *Turtle geometry* de Harold ABELSON et Andrea DISESSA paru aux MIT Press en...1981.

1 Et le taupin créa la tortue

Nous allons créer notre exploratrice mathématique à partir des commandes les plus simples : « tourne », « avance », « saute »...
Pour cela, nous allons, d'un point de vue informatique, utiliser ce qu'il faut en d'autres circonstances à tout prix éviter : des *variables globales*.
En effet, notre tortue sera définie par trois paramètres pour assurer l'interface avec Maple :

— **pos** qui contiendra les coordonnées de la tortue sous forme d'une liste **[abs,ord]** pour que Maple puisse la repérer ;

— **dir** qui contiendra un entier représentant la direction que suivra la tortue pour se déplacer par rapport à sa direction de départ. Cette angle sera exprimé en degré ;

— **chemin** sera un objet graphique Maple qui permettra d'afficher tous les déplacements de la tortue.

Nous partirons donc de :

Maple

```
> pos := [0,0]:
  dir := 0.:
  chemin := plottools[point](pos, color = pink, symbol = box):
```

La géométrie va alors illustrer la notion informatique d'état d'une variable et inversement, notion qu'on retrouve d'ailleurs en physique et en chimie.
Voici les principales commandes qui nous permettront de partir en exploration. Analysez-les et comprenez-les et améliorez-les si besoin...

Maple

```
> tourne := proc(angle)
    global dir,chemin;
    dir := dir + evalf(Pi)*angle/180.;
  end:
> avance := proc(long)
    global pos,dir,chemin;
    local pos0;
    pos0 := pos;
    pos := pos + [long*cos(dir),long*sin(dir)];
    chemin := chemin,plot([pos0,pos]);
  end:
> saute := proc(long)
    # permet à la tortue d'avancer de long dans la direction dir sans
        laisser de trace...
> efface := proc()
    # permet de ré-initialiser le chemin
> affiche := proc()
    global chemin;
    chemin := chemin, plottools[point](pos, color = red, symbol = diamond):
    plots[display](chemin, scaling = constrained, axes = none);
  end:
```

Expérimentez pour bien comprendre le rôle et les spécificités de chaque procédure.

2 Polygone

Our Women are Straight Lines.

Our Soldiers and Lowest Class of Workmen are Triangles with two equal sides, each about eleven inches long, and a base or third side so short (often not exceeding half an inch) that they form at their vertices a very sharp and formidable angle. Indeed when their bases are of the most degraded type (not more than the eighth part of an inch in size), they can hardly be distinguished from Straight lines or Women ; so extremely pointed are their vertices. With us, as with you, these Triangles are distinguished from others by being called Isosceles ; and by this name I shall refer to them in the following pages.

Our Middle Class consists of Equilateral or Equal-Sided Triangles.

Our Professional Men and Gentlemen are Squares (to which class I myself belong) and Five-Sided Figures or Pentagons.

Next above these come the Nobility, of whom there are several degrees, beginning at Six-Sided Figures, or Hexagons, and from thence rising in the number of their sides till they receive the honourable title of Polygonal, or many-Sided. Finally when the number of the sides becomes so numerous, and the sides themselves so small, that the figure cannot be distinguished from a circle, he is included in the Circular or Priestly order ; and this is the highest class of all.

in « Flatland : A Romance of Many Dimensions » de Edwin Abbott (1884)

2.1 Des expériences

Nous voici prêts : en utilisant uniquement les commandes précédentes, commencer par tracer un triangle équilatéral, un carré , puis une procédure `triangle := proc(cote)` qui va tracer un triangle équilatéral de côté `cote`. On peut faire de même pour les carrés avec une procédure `carre := proc(cote)`.

Bon, euh, c'est en traçant des carrés avec une tortue que vous allez rentrer à Ulm ?...

Arrêtons-nous un instant sur la nature de la géométrie de la tortue en la comparant à la géométrie cartésienne : quelles différences voyez-vous entre les deux ?

La différence est encore plus nette lorsqu'il s'agit de tracer un cercle. En géométrie cartésienne, on utilisera l'équation $x^2 + y^2 = R^2$.

Comment tracer un cercle avec la tortue ? Quelles notions mathématiques se cachent derrière ?

Que se passe-t-il avec l'objet décrit par $x^2 + 2y^2 = R^2$?

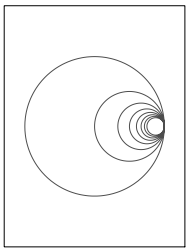
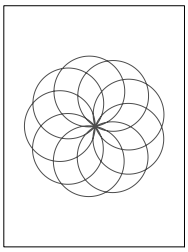
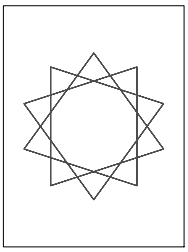
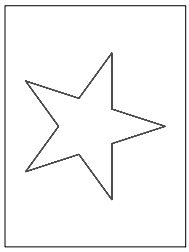
La tortue ne s'occupe donc que des propriétés intrinsèques des figures.

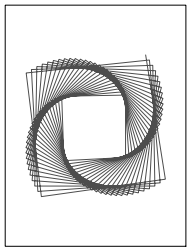
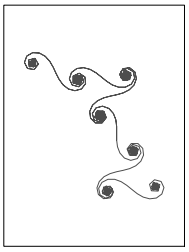
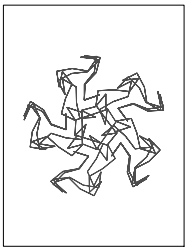
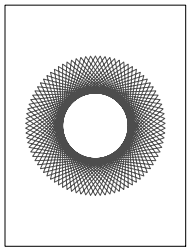
De plus elle n'a de vision que locale : elle n'a pas de vision globale de l'objet qu'elle explore. Cela peut paraître un inconvénient mais c'est en fait une force qui lui permet d'aborder des notions mathématiques bien plus poussées, notamment dans l'exploration des surfaces.

Enfin la tortue utilise des procédures et échappe au formalisme des équations algébriques ce qui permet d'aller plus loin avec des outils simples.

Bon, passons à la récréation...

Dessinez des petites choses comme ça :

avec des procédures du type :

Maple

```
> poly1 := proc(cote,angle,nb) # la tortue avance nb de cote unités et
    tourne d'angle degrés
> poly2 := proc(cote,angle,facteur,nb) # on tourne tantôt d'angle, tantôt
    d'angle*facteur
> poly_spirale := proc(cote,angle,inc,nb) # procédure récursive qui
    incrémente le côté
> poly_spirale2 := proc(cote,angle,inc,nb) # procédure récursive qui
    incrémente l'angle
```

Peut-on contrôler le rayon d'un cercle tracé par la tortue ?
Déterminez une procédure qui dessine un arc de cercle.
Le rayon du cercle est une notion globale qui est malgré tout liée à une quantité locale que la tortue peut donc mesurer.

2 2 Un théorème

Observez vos **poly1** : semble-t-il y avoir une relation entre le nombre de côtés et l'angle ?
On peut distinguer deux classes de **poly1** par leur « aspect » : est-ce que cela se retrouve dans la relation précédente ?
Que se passe-t-il pour des chemins clos quelconques ? On dira qu'un chemin clos est simple s'il n'a pas de boucle.
Comment peut-on alors retrouver un vieux théorème sur les angles intérieurs d'un triangle ? Et pour des polygones « sans boucle » quelconques ?
Où se retrouvent les sommets de n'importe quel **poly1** ? Pourriez-vous le démontrer ?
Démontrez alors le théorème suivant :

Théorème 2 - 1

Un chemin tracé par la procédure **poly1** se refermera exactement lorsque la rotation totale atteindra un multiple de 360 ° .

Écrivez alors une procédure **poly** qui s'arrête quand le chemin se ferme.

3 Une tortue prédatrice

Notre tortue gagne en confiance et part s'amuser à Las Vegas : comme vous le savez, de nombreuses méthodes de recherche sont plus efficaces si les éléments de l'ensemble exploré sont choisis aléatoirement.
On va tout d'abord observer une tortue se déplaçant aléatoirement : nous allons faire suivre à **avance** et **tourne** une loi uniforme en utilisant la fonction **rand** de Maple.

Maple

```
> rand(1..6);
proc()
local t;
global _seed;
```

```
    _seed := irem(427419669081*_seed, 999999999989);
    t := _seed;
    irem(t, 6) + 1
end
> rand(1..6)();
                                4
```

Créez alors une procédure :

Maple

```
random_move := proc(d1,d2,a1,a2,n)
```

avec d1 et d2 les bornes de la direction, a1 et a2 les bornes de la longueur des pas et n le nombre de pas.
Faites varier les paramètres. Des conjectures ? Comment étudier plus scientifiquement tout ceci ?
Bon, dotons à présent notre tortue d'un odorat et utilisons-la comme chasseuse...
On va dire que l'odeur sera inversement proportionnelle à la distance de la proie. On crée une proie et une fonction qui calcule la distance.

Maple

```
> proie := [200,200]:
> distance := proc(c1,c2)
   RETURN(((c2[1]-c1[1])^2 + (c2[2]-c1[2])^2))
  end:
```

On va dire qu'à chaque étape, la tortue va avancer d'une distance aléatoire et tourner la tête aléatoirement. Si elle sent qu'elle s'éloigne, elle va se tourner d'un angle fixe et recommencer à avancer. Sinon, elle recommence à avancer.
Vous aurez donc une procédure :

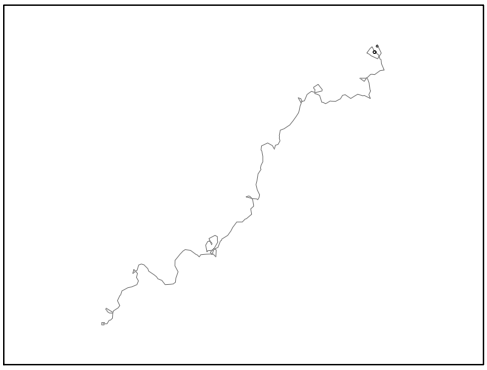
Maple

```
> chasse := proc(d,a,o,n)
```

sachant que la tortue avance d'une distance uniformément choisie dans $\{1, 2, .., a\}$, tourne aléatoirement sa tête d'un angle compris entre $-d$ et d pour chercher l'odeur et si elle sent qu'elle s'éloigne, elle tournera toujours la tête de o et ce pendant n étapes.
Par exemple :

Maple

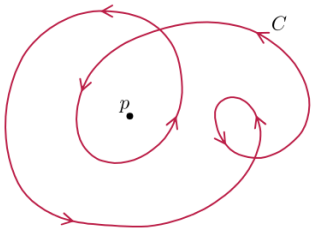
```
> efface():chasse(60,5,90,200):affiche();
```



On peut améliorer le modèle en dotant la tortue de deux narines : selon l'odeur reçue dans chaque narine, elle ira plutôt vers la droite ou la gauche.
On peut ensuite faire bouger la proie qui peut être elle aussi dotée d'un odorat et s'éloignera de sa prédatrice.
Ensuite, il faudra étudier les probas, la dynamique derrière tout ça...tout un sujet de TIPE ;-)

4 Une tortue topologique

La tortue, en plus de dynamiser la géométrie va aussi la faire fondre. Nous allons en effet travailler à présent sur des objets mous, disons en pâte à modeler.
Dans ce monde, un carré, un triangle, un cercle sont en fait un même objet : quelque chose qui a la même « forme » : pour nous, ce sera le même nombre de « tours » pour parcourir le chemin.
Prenez un triangle, vous le « déformez » petit à petit pour qu'il devienne un carré. Pour le parcourir, la tortue fait toujours un seul tour : on dit alors que le nombre de tours de la tortue (on va l'appeler *indice chélonien* ($\chi\varepsilon\lambda\omega\nu\eta$ voulant dire tortue...)) est un *invariant topologique*.
Lorsque vous effectuez ces « petites » déformations, vous ne pouvez changer l'indice chélonien que faiblement or cet indice est un entier donc il ne va pas bouger.
Bon, c'est un peu intuitif et vasouilleux : peut-on faire mieux sans attendre d'être en Master 2 ?



Il y a des problèmes : on peut faire disparaître une boucle par « petites déformations », on peut en créer trois d'un coup, et en parcourant un cercle dans l'autre sens, on ne déforme rien mais dans tous les cas on change l'indice chélonien tout d'un coup.
Alors c'est quoi une « petite déformation » ? On va dire que c'est une déformation qui ne change pas la procédure de parcours du chemin.
En fait, il faut penser à un chemin comme étant un programme pour tortue (dynamique) et non pas un dessin (statique). . Si vous gardez ça en mémoire quelques années, vous aurez peut-être plus de facilité à aborder la topologie...
Ainsi, ce qui sur le dessin paraît un petit changement en est un énorme dans le programme (il faut faire faire tout d'un coup un tour de plus sur elle-même à la tortue).
Bon, puisqu'il faut regarder les programmes qui font bouger la tortue, il faudrait un peu les normaliser pour les comparer.
Premier problème : `tourne(90)` et `tourne(-270)`. Pour y remédier, on conviendra de n'utiliser que des mesures principales pour les angles (de valeur absolue inférieure à 180).
Ceci nous permet par exemple de pouvoir déceler les petites déformations du dessin qui font disparaître une boucle [a] : **comment ?**
Bon, nous voulons bien croire que deux chemins de même type topologique ont le même coefficient chélonien mais qu'en est-il de la réciproque. Et bien c'est encore vrai et cela

a. Est-ce que M. Sauvageot vous a déjà fait le coup du nœud de cravate ?

constitue un beau théorème publié en 1937 :

Théorème 2 - 2

Théorème de Whitney-Graustein
Deux chemins fermés du plan peuvent être déformés de l'un vers l'autre si, et seulement si, ils ont le même coefficient chélonien.

Bon, cela n'a pas été écrit comme ça...Vous pouvez lire le court article publié par Whitney à cette adresse :
http://archive.numdam.org/article/CM_1937__4__276_0.pdf
et celui-ci paru vingt ans plus tard sous la plume de John Griffin :
http://archive.numdam.org/article/CM_1956-1958__13__270_0.pdf
Enfin, vous pourrez regarder cette preuve moderne donnée en 2007 par Hansjörg Geiges :
http://arxiv.org/abs/0801.0046
Bon, notre petite tortue nous fait voyager dans une mathématique plutôt moderne...
On pourrait avoir une autre idée « statique » : le nombre de points d'intersection dans le chemin pourrait nous donner aussi un invariant topologique : qu'en pensez-vous ? Quels sont les coefficients chéloniens possibles de chemins ayant un seul point d'intersection ? Deux ? Trois ?

5 Le tour du monde de la tortue

Une tortue part de l'Équateur, avec un angle de 90°. Elle arrive au Pôle Nord et tourne à nouveau de 90°. Elle retourne sur l'Équateur, retourne de 90° : elle revient donc à son point de départ. Elle a donc suivi un chemin fermé et elle a tourné de...270° ! Voilà qui met à mal notre théorème initial.
De plus, si elle tourne de 70° au Pôle Nord, elle aura tourné au total de 250°... Notre coefficient chélonien n'est plus entier ?!

Les déplacements de la tortue sur un plan étaient régis par des **avance**, donc des petits bouts de ligne droite mais au fait, c'est quoi une ligne droite sur une sphère ? Pour que la tortue sache qu'elle va en ligne droite, il faudrait que la caractérisation d'une ligne droite soit locale...
Marcher tout droit, sans tourner, qu'est-ce que c'est ? Essayez de marcher « tout droit » le long d'un parallèle différent de l'Équateur.
Autre problème : la tortue marche « tout droit » d'un Pôle à l'autre (sans utiliser de **tourne**) puis revient au Pôle initial en faisant des pas de côté : elle n'a pas tourné la tête et pourtant que se passe-t-il ?
Maintenant, revenons au grand triangle de 270°. Imaginons que la tortue garde avec elle un pointeur indiquant sa direction initiale : elle peut faire ça localement avec un peu de mémoire. À chaque fois qu'elle tourne, elle laisse le pointeur vers la direction initiale : que se passe-t-il pour le pointeur quand elle revient au départ ?

TP N°

3 Courbes elliptiques

En 1995, le mathématicien anglais Andrew Wiles fit sensation en démontrant enfin le grand théorème de Fermat, trois siècles après sa formulation, à l'aide des courbes elliptiques ce qui surprit les spécialistes de la théorie des nombres. Elles jouent actuellement un rôle essentiel dans la sécurité informatique mais on les retrouve en géométrie, en mécanique. Elles vont nous permettre aujourd'hui de mettre un pied dans un domaine mathématique encore tout chaud...

1 Un peu de lecture...

Nous ne pouvons, à notre niveau, que parcourir rapidement ce riche domaine des mathématiques en admettant la plupart des résultats que vous démontrerez peut-être un jour. Pour les plus curieux, vous pourrez lire le très bel ouvrage de Marc Hindry *Arithmétique : Primalité et codes, Théorie analytique des nombres, Equations diophantiennes, Courbes elliptiques* paru en 2008 chez Calvage & Mounet ainsi que l'*Introduction élémentaire à la théorie des courbes elliptiques*[a] de Marc Joye de l'Université de Louvain[b], un des hauts lieux de la cryptographie, où vous retrouverez de nombreuses démonstrations.
Les ouvrages suivants proposent des activités pratiques :

— *Initiation à la cryptographie* de Gilles Dubernet (Vuibert 2002) ;

— *Algèbre, arithmétique et maple* de Bernadette Perrin-Riou (Cassini 2000) ;

— *Mathématiques & informatique* de François Morain et Jean-Louis Nicolas (Vuibert 1995).

2 Quelques résultats préliminaires

La méthode de la tangente et de la sécante dont nous allons parler n'est pas nouvelle. Fermat, voire Diophante, ont tourné autour puis Newton, Lagrange, Cauchy, Sylvester, Poincaré. Il faut pourtant attendre les années 1920 et André Weil pour qu'elle soit parfaitement maîtrisée même si un article publié en allemand en 1896 par le Danois Juel donne une description complète de la loi de groupe sur une cubique mais semble ne pas avoir été lu par ses contemporains[c]...
Il a cependant fallu attendre la fin du XX^{e} siècle pour l'utiliser en arithmétique avec le succès que l'on sait car ces courbes permettent d'obtenir efficacement des factorisations d'entiers comme le montra le néerlandais Hendrik Lenstra.
Nous aurons besoin de connaître quelques résultats sur les groupes et les corps finis que vous avez découvert avec M. Sauvageot[d].
Nous aurons également besoin de parler de manière intuitive du plan projectif $\mathbb{P}_2(\boldsymbol{K})$ que vous avez découvert lors du chapitre sur les quadriques et les polynômes homogènes sur $\boldsymbol{K}[X,Y,Z]$.
Disons qu'en gros, on définit sur $\boldsymbol{K}^3 \setminus \{(0,0,0)\}$ une relation d'équivalence :

$$(X,Y,Z) \equiv (X',Y',Z') \Leftrightarrow \exists t \in \boldsymbol{K}^*,\ (X',Y',Z') = t(X,Y,Z)$$

et que $\mathbb{P}_2(\boldsymbol{K})$ est l'ensemble des classes d'équivalence.
On établit une « correspondance » entre l'espace affine $\mathbb{A}_2(\boldsymbol{K})$ et $\mathbb{P}_2(\boldsymbol{K})$: au point de coordonnées (x,y) du plan affine, on fait correspondre le point de coordonnées $(X:Y:Z)$ dans le plan projectif en posant $x = \frac{X}{Z}$ et $y = \frac{Y}{Z}$. Se pose alors le problème du cas $Z = 0$. Ce cas correspond au « point à l'infini » où deux droites parallèles du plan projectif se croisent. Nous en distinguerons par la suite un particulier de coordonnées $(0:1:0)$ que nous noterons $\mathcal{O}$.

a. http://sciences.ows.ch/mathematiques/CourbesElliptiques.pdf
b. http://www.uclouvain.be/crypto/publications/latest
c. Voir l'article de Norbert Schappacher www-irma.u-strasbg.fr/ schappa/NSch/Publications_files/DPP.pdf
d. Qui, comme vous le savez, est le traducteur d'une des « bibles » de la théorie des nombres : *Introduction à la théorie des nombres* paru en 2006 chez Vuibert...

3 Loi de groupe sur une cubique

On peut montrer que dans le plan projectif, toute droite coupe une cubique d'équation $Y^2Z = X^3 + aXZ^2 + bZ^3$ en exactement trois points, en tenant compte de l'ordre de multiplicité, lorsque a et b remplissent certaines conditions qui rendent la courbe « lisse » et sans racine double.
En fait, quand on repasse « en affine », on obtient $y^2 = x^3 + ax + b$. Vous avez évoqué le discriminant de $x^3 + ax + b$ lors de l'étude de la méthode de CARDAN pour résoudre les équations de degré 3. En fait, il faut que ce discriminant soit non nul : nous supposerons donc par la suite que $4a^3 + 27b^2 \neq 0$.
Il faut aussi que la caractéristique du corps soit différente de 2 ou 3 mais cela ne nous inquiète pas car nous allons travailler dans des corps finis de très grande caractéristique (factoriser par des petits nombres ne nécessite pas de sortir l'artillerie lourde...).
Bon, regardons les points à l'infini : $Z = 0 \Longrightarrow X = 0$. Il n'y a donc qu'un seul qui est dans la direction $x = 0$ et qui a pour coordonnées dans $\mathbb{P}_2$ $\mathcal{O}(0 : 1 : 0)$.

Par la suite, nous considérerons donc une cubique $\mathcal{E}$ d'équation $y^2 = x^3 + ax + b$ de discriminant non nul.
On notera $\mathcal{E}(\boldsymbol{K})$ les solutions (x, y) de l'équation $y^2 = x^3 + ax + b$ sur $\boldsymbol{K}^2$ auxquelles on ajoute $\mathcal{O}$.
Par exemple, on peut visualiser ce que cela donne sur $\mathbb{R}$:

```
> anim_elliptic:=proc()
   local P,a,b;
   P:=NULL;
   for a from -10 to 5 by 2 do
     for b from -10 to 5 by 2  do
       if (4*a^3+27*b^2)<>0 then
        P:=P,plots[implicitplot](y^2=x^3+a*x+b,x=-10..10,y=-10..10);
   fi;od;od;
   plots[display]([P],insequence=true,view=[-10..10,-10..10]);
  end:
```

Soit P un point quelconque. Une droite $(P\mathcal{O})$ est donc une droite passant par P et parallèle à l'axe $x = 0$.
On a admis que toute droite coupait $\mathcal{E}$ en un troisième point en tenant compte de la multiplicité. On peut donc définir une loi sur $\mathcal{E}(\boldsymbol{K})$:

- si P et Q sont deux points distincts de $\mathcal{E}(\boldsymbol{K})$, alors la droite (PQ) recoupe la courbe en un unique troisième point que nous noterons $P \star Q$. La droite joignant $P \star Q$ et $\mathcal{O}$ recoupe la courbe en un troisième point que nous noterons $P + Q$.
- si $P = Q$, le point $P \star P$ sera l'intersection de la tangente avec la courbe. On admettra (ce sera vu plus tard dans l'année avec M. Sauvageot) qu'une courbe lisse admettant une équation implicite de la forme $f(x, y) = 0$ admet une tangente au point (x_0, y_0) d'équation :

$$(x - x_0)\frac{\partial f}{\partial x}(x_0, y_0) + (y - y_0)\frac{\partial f}{\partial y}(x_0, y_0) = 0$$

On peut vérifier que cette loi est commutative, que $\mathcal{O}$ en est l'élément neutre. et que le symétrique de P par rapport à l'axe $(0x)$ est le symétrique de P pour la loi $+$ et qu'il appartient à la courbe.
Il resterait à vérifier l'associativité mais cela nécessiterait pas mal de travail et nous n'en avons pas le temps : vous pourrez étudier la littérature pour vous en convaincre.

1. Soit $P_1(x_1, y_1)$ et $P_2(x_2, y_2)$. Déterminez les coordonnées (x_3, y_3) de la somme $P_1 + P_2$ en distinguant les différents cas. On prendra $\boldsymbol{K = R}$

   Déduisez-en une procédure `somme := proc(P1,P2,a,b)` qui calcule les coordonnées de la somme $P_1 + P_2$ pour une courbe elliptique donnée dépendant des paramètres a et b. On pourra créer une procédure intermédiaire qui vérifie qu'un point appartient à la courbe elliptique de paramètres a et b. On symbolisera le point $\mathcal{O}$ par `[0]` (à ne pas confondre avec `0` ...).

   Vous pourrez ensuite visualiser la construction :

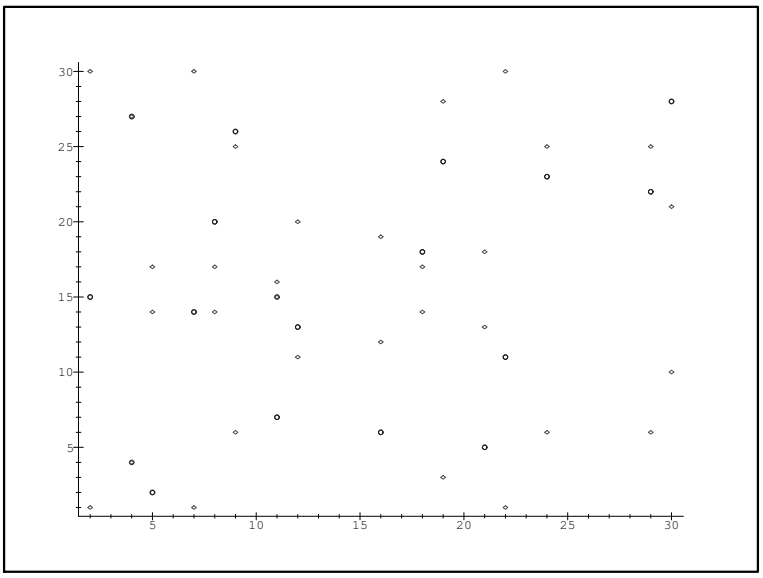
```
> dessin_elliptic:=proc(P1,P2,a,b)
    local P3,P4,y2,E,L1;
    P3 := somme(P1,P2,a,b);
    if P1 = [0] or P2 = [0] or P3 = [0]
       then ERROR('Point à l infini');fi;
    P4 := [P3[1],-P3[2]];
    E := plots[implicitplot](y^2 = x^3 + a*x+b,x=-10..10,y=-10..10
        ,color=green);
    L1 := plot([P1,P4,P3],color=blue,thickness=3);
    plots[display]([E,L1]);
  end:

> point_courbe:=proc(x,a,b)
    RETURN([(x),(sqrt(x^3+a*x+b))]);
  end:

> dessin_elliptic(point_courbe(-2,-5,3),point_courbe(-1,-5,3),-5,3);
```

2. On veut maintenant tracer la courbe en travaillant dans $\mathbb{F}_p$ avec bien sûr p un nombre premier. Qu'est-ce qui va changer dans nos procédures ?

   La recherche d'une racine carrée dans $\boldsymbol{Z}/n\boldsymbol{Z}$ n'est pas tout à fait évidente. Pendant vos temps libres, vous pourrez vérifier que si p est un nombre premier congru à 3 modulo 4, alors on arrive à s'en sortir mais si p est congru à 1, alors on passe par l'algorithme probabiliste de SHANK. Nous n'avons pas besoin d'être efficace ici car il s'agit seulement de dessiner la courbe avec une valeur de p petite donc une recherche exhaustive suffira.

   Pour ce qui est de la recherche de l'inverse, on a déjà traité ce cas dans les TD précédents et Maple sait d'ailleurs se débrouiller : `1/16 mod 29` renvoie bien 20 car $16 \times 20 \equiv 1[29]$ mais `1/15 mod 27` renvoie une erreur.

   Voici par exemple la courbe elliptique dans $\boldsymbol{Z}/31\boldsymbol{Z}$ d'équation $y^2 = x^3 + 26x + 3$ où les points sont représentés par des losanges ainsi qu'une droite passant par deux points de la courbe dont les points ont été représentés par des cercles...



Voilà qui perturbe un peu nos habitudes !...

3. Que se passe-t-il dans $\mathbb{Z}/n\mathbb{Z}$ lorsque n est composé ? Nous allons voir plus loin que ce n'est pas un problème et qu'il s'agit plutôt de la clé du succès des courbes elliptiques.
4. Déterminez une procédure donnant les différents multiples d'un point dans le groupe associé à une courbe elliptique donnée.

```
> multiples(point_courbe_p(1,26,3,31),33,26,3,31);

[0], [2, 1], [16, 12], [22, 1], [7, 30], [11, 15], [12, 11],
[18, 14],[5, 14], [29, 6], [19, 28], [24, 6], [21, 13], [9, 6],
[30, 10],[8, 17], [4, 4], [4, 27], [8, 14], [30, 21], [9, 25],
[21, 18],[24, 25], [19, 3], [29, 25], [5, 17], [18, 17],
[12, 20],[11, 16], [7, 1], [22, 30], [16, 19], [2, 30], [0]
```

5. Déterminez une procédure `k_multiple_p:=proc(P,k,a,b,p)` qui calcule le k^{e} multiple du point P. Vous pourrez améliorer votre procédure en utilisant un algorithme d'exponentiation rapide en pensant à la décomposition en base 2 de k.

4 Factorisation d'entiers : méthode de Lenstra

1. Nous cherchons à factoriser un entier impair donné que nous savons composé (par exemple pour casser une clé RSA).
   Dans notre procédure de calcul de somme dans $\mathbb{Z}/n\mathbb{Z}$, des divisions peuvent ne pas être effectuées. Maple renvoie alors un message d'erreur :

```
> 7/3 mod 21;
'Error, the modular inverse does not exist'
```

   Pourquoi ? Si tel est le cas, que pouvons-nous dire de la factorisation de n ? Comment modifier alors la procédure d'addition sur $\mathcal{E}(\mathbb{Z}/n\mathbb{Z})$ pour obtenir un facteur de n ?
2. On dit qu'un nombre n est B-super friable si toutes les puissances entières de nombres premiers divisant n sont inférieures à B.
   Par exemple, $2^4 \times 3^2 \times 11$ est 16-super friable.

3. Si le mathématicien allemand Helmut HASSE n'a pas été très inspiré dans ses choix politiques en soutenant le régime nazi[e], il a été un très grand algébriste et théoricien des nombres. Il a en particulier démontré le théorème suivant :

$$|\#\mathcal{E}(\mathbb{F}_p) - (p+1)| \leqslant 2\sqrt{p}$$

Sa démonstration fait référence à la fonction ζ de RIEMANN qui est a priori un outil d'analyse mais qui a eu une énorme influence en théorie des nombres comme nous le verrons dans un prochain TP.

4. Soit p un diviseur premier de n. On suppose que le cardinal c_p de $\mathcal{E}(\mathbb{F}_p)$ est B-super friable. Soit P un point de $\mathcal{E}(\mathbb{Z}/n\mathbb{Z})$ différent de $\mathcal{O}$ et soit k le PPCM de $(1, 2, .., B)$. Comme c_p divise k et que $P \in \mathcal{E}(\mathbb{F}_p)$ car p divise n, alors, d'après le théorème de LAGRANGE, on sait que $kP = \mathcal{O}$ dans $\mathcal{E}(\mathbb{F}_p)$.

   Cependant, on ne connaît pas p donc on va calculer kP dans $\mathcal{E}(\mathbb{Z}/n\mathbb{Z})$. Or des problèmes peuvent apparaître car il va y avoir des diviseurs de 0 : le point précédent nous permet de trouver un facteur de n. Si c'est un facteur trivial, on choisit une autre courbe. Sinon, on a un diviseur de n ! Si le calcul de kP ne pose pas de problème, on choisit un autre point P ou une autre courbe elliptique.

   Le problème est dans le choix du bon point et de la bonne courbe...

   Voici un exemple proposé par Marc JOYE.

   On veut factoriser $n = 540143$ par la méthode de LENSTRA. On va travailler sur les courbes elliptiques

$$\mathcal{E}_a : y^2 = x^3 + ax + 1$$

   Le point $P(0, 1) \neq \mathcal{O}$ appartient toujours à $\mathcal{E}_a$. Prenons $a = 1$. On vérifie que le discriminant est non nul modulo n.

   On sait qu'un diviseur premier p de n est majoré par $\sqrt{n} \approx 735$

   Le nombre de points de la courbe est majoré, d'après le théorème de HASSE, par $735 + 1 + 2\sqrt{735} < 792$. On choisit donc $B = 792$.

   On choisit un nombre 792-super friable, par exemple $k = 2^9 \times 3^6$.

   Si le calcul n'aboutit pas, on a gagné, sinon, on augmente a de 1 ou on augmente B (par exemple $B = 2^9 \times 3^6 \times 5^4$).

   Il a été montré que la complexité de cet algorithme est en $\mathcal{O}\left(e^{(1+\varepsilon)\log p \log\log p}\right)$, où p est le plus petit facteur premier de n : c'est bien plus efficace que les méthodes classiques.

   Aidez-vous de Maple pour factoriser n avec cette méthode.

```
> facto_p(nextprime(123456789)*nextprime(98765432));

12193264407559831 = 98765441 * 123456791 avec a = 49
```

   On notera cependant qu'un concurrent est arrivé récemment et a remisé la méthode de factorisation par courbes elliptiques au deuxième rang des méthodes de factorisation les plus efficaces : il s'agit du crible algébrique dont la complexité est en $\mathcal{O}\left(e^{\left(\frac{64}{9}\log n\right)^{\frac{1}{3}}(\log\log n)^{\frac{2}{3}}}\right)$.

   Cependant, la méthode « elliptique » est plus rapide pour trouver de petits facteurs.

e. Il avait pourtant un nom de famille pouvant plaire aux nazis mais ceux-ci ont toujours refusé son adhésion au parti car on le suspectait d'avoir des origines juives...

5 Cryptographie

En fait, on reprend des méthodes introduites ces dernières années (El Gamal) sur $(\mathbb{F}_p^\star, \times)$ mais en travaillant sur des courbes elliptiques. La difficulté tient au fait qu'il est difficile de calculer un logarithme discret. Or les groupes définis sur des courbes elliptiques sont très divers, on ne sait même pas trouver rapidement leur ordre et encore moins leur structure : le problème du logarithme discret devient beaucoup plus compliqué et inversement les calculs du codeur se font très rapidement sur des courbes elliptiques définies sur $\mathbb{F}_{2^m}$.

Par exemple, on estime actuellement qu'une clé de 256 bits avec des courbes elliptiques assure la même sécurité qu'une clé de 3072 bits avec RSA...

Cela permet donc d'assurer une grande sécurité sur les petits processeurs des cartes à puces.

Un problème extra-mathématique existe cependant : les algorithmes liés à ces méthodes sont cachés sous d'horribles brevets qui rend le coût de leur utilisation exorbitant...

6 Jokers

Si vous éprouvez quelques difficultés...

```
> proc1 := proc(x,p)
  local i,r;
  r:=NULL;
  for i from 0 to p-1 do
    if i^2 mod p = x
    then r := r,i;
    fi;
  od;
  RETURN([r]);
  end:
```

```
> proc2 := proc(P1,P2,a,b,p)
   local x1,x2,y1,y2,x3,y3,lambda;
   if (4*a^3+27*b^2) mod p = 0  then ERROR(‘Delta nul‘);fi;
   if  (P1=[0] or appart_p(a,b,P1,p)) and (P2=[0] or
       appart_p(a,b,P2,p))   then
     if P1 = [0] then RETURN(P2); fi;
     if P2 = [0] then RETURN(P1); fi;
     x1:=P1[1] mod p;y1:=P1[2] mod p;x2:=P2[1] mod p;y2:=P2[2] mod p;
     if x1 = x2 mod p and y1 = -y2 mod p then RETURN([0]);fi;
     if P1 = P2 then
        if y1 = 0 mod p then RETURN([0]);fi;
           if igcd(y1,p) <  1 then RETURN([‘Diviseur‘,igcd(y1,p)]);fi;
        lambda := (3*x1^2+a)/(2*y1) mod p;
     else
        if igcd(x1-x2,p) <  1 then
            RETURN([‘Diviseur‘,igcd(x1-x2,p)]);fi;
        lambda := (y1-y2)/(x1-x2) mod p;
     fi;
     x3 := (lambda^2-x1-x2) mod p;
     y3 := (-y1 +lambda*(x1-x3)) mod p;
     RETURN([x3,y3]):
   else ERROR(‘Mauvais points‘);
   fi;
 end:
```

```
> proc3 := proc(x,a,b,p)
    local X;
    X := x;
    while sqrt_p((X^3+a*X+b) mod p,p) = [] and X < p do
      X := X+1;
    od;
    RETURN([X,sqrt_p((X^3+a*X+b) mod p,p)[1]]);
  end:
```

```
> proc4 := proc(x1,x2,a,b,p)
    local y2,E,x,i,P1,P2,P,lambda;
    if (4*a^3+27*b^2) mod p = 0 then ERROR(‘Delta nul‘);fi;
    P1 := point_courbe_p(x1,a,b,p);
    P2 := point_courbe_p(x2,a,b,p);
    E := []; P:= [P1,P2];
    lambda := (P2[2]-P1[2])/(P2[1]-P1[1]) mod p;
    for x from 0 to p-1 do
```

```
    y2 := (x^3 + a*x + b) mod p;
    for i in sqrt_p(y2,p) do
      E := [op(E),[x,i]];
      P := [op(P),[x,(lambda*(x-P1[1])-P1[2]) mod p]];
    od;
  od;
  E := plot(E,style=point,symbol=diamond,color=red);
  P := plot(P,style=point,symbol=circle,color=blue);
  plots[display]([E,P]);
 end:
```

```
> proc5 := proc(P,k,a,b,p)
  local K,M,Q,r;
  K:=k; M:=[O]; Q := P mod p;
  while K   0 do
    r := K mod 2;
    if r = 1 then M := somme_p(M,Q,a,b,p); fi;
    if member('Diviseur',M) then RETURN(M);fi;
    Q := somme_p(Q,Q,a,b,p);
    if member('Diviseur',Q) then RETURN(Q);fi;
    K := iquo(K,2);
    od;
  RETURN(M);
  end:
```

```
> proc6 := proc(n)
  local deux,trois,sept,onze,h;
  deux:=2;trois:=3;sept:=7;onze:=11;
  h:=rand(n);
  while deux <= h() do deux := deux*2;od;
  while trois <= h() do trois := trois*3;od;
  while sept <= h() do sept := sept*7;od;
  while onze <= h() do onze := onze*11;od;
  RETURN(deux*trois*sept*onze/462);
  end:
```

```
> proc7 := proc(n)
  local kP,a,b,P,B,s,k;
  if isprime(n) then ERROR('premier!');fi;
  P:=[0,1];
  s := ceil(sqrt(n)):
  B:=s+1+2*ceil(sqrt(s));
  k:=friable(B);
  a:=1;b:=1;
  kP:=k_multiple_p(P,k,a,b,n);
  while not member('Diviseur',kP) do
    a := a+1; # on peut choisir de jouer aussi sur B
    # k := friable(B);
    kP:=k_multiple_p(P,k,a,b,n);
    od;
  printf('%a = %a * %a avec a = %a',n,kP[2],n/kP[2],a);
  end:
```