ALGORITHMIQUE

Un algorithme est une succession d'instructions (on dit aussi « ordres » ou « commandes ») dont l'exécution par un ordinateur permet de résoudre un problème.

L'algorithme conçu, on le traduit dans un « langage » spécifique (par exemple, basic, Maple, Caml, Pascal ou le langage utilisé par les calculatrices) permettant de programmer l'ordinateur. Ce dernier convertit alors les instructions dans son propre langage de base : le langage machine.

Le principe fondamental de l'algorithmique est de décomposer un problème complexe en des problèmes de plus en plus simples jusqu'à faire apparaître des actions élémentaires que connaît l'ordinateur. Ces actions élémentaires sont :

- **Affecter une valeur à une mémoire.**
- **Lire une valeur dans une mémoire.**
- **Additionner, soustraire, multiplier, diviser le contenu de deux mémoires.**
- **Comparer le contenu de deux mémoires.**

Les instructions élémentaires disponibles sont donc en nombre très limité. Il s'agit de l'affectation, des opérations d'entrée/sortie (permettant les entrées de données et les affichages de résultats), les tests et les boucles. On peut écrire tous les algorithmes, même les plus complexes, avec ces quelques instructions.

Une ***proposition*** ou ***condition*** (termes qui apparaîtront dans les définitions suivantes) est une affirmation (la plupart du temps une égalité ou une inégalité) qui est soit vraie, soit fausse.

Les algorithmes seront écrits en italique. Il est possible d'écrire plusieurs instructions sur une même ligne. Dans ce cas, on les séparera par un point virgule.

L'affectation

C'est l'instruction qui permet d'affecter une ***valeur*** à une ***variable***. Une variable est en fait une zone mémoire dans laquelle on peut stocker une valeur afin de l'utiliser ultérieurement. Il existe différents types de valeurs : numériques, alphanumériques (texte), booléennes (ne pouvant prendre que deux valeurs : vrai ou faux), etc.

Syntaxe 1 : *var* $\leftarrow$ *val ;* $\Rightarrow$ Affecte la valeur *val* à la variable *var*.
Ex : *b* $\leftarrow$ 3 *;* (*b* est une variable, 3 est une valeur).

Syntaxe 2 : $var_1 \leftarrow var_2$ *;* $\Rightarrow$ Affecte la valeur de la variable var_2 à la variable var_1.
Ex : *c* $\leftarrow$ *b ;* (c'est la valeur de *b*, c'est-à-dire 3, qui est affectée à la variable *c*).

Exemple 1 : Echanger le contenu de deux variables *a* et *b*.
c $\leftarrow$ *a ; a* $\leftarrow$ *b ; b* $\leftarrow$ *c ;*
Commentaire : Une troisième variable *c* est nécessaire. En effet, *a* $\leftarrow$ *b ; b* $\leftarrow$ *a ;* conduirait à obtenir la valeur initiale de *b* à la fois dans *a* et dans *b* puisque le contenu initial de *a* est perdu dès la première action lorsqu'il est remplacé par celui de *b*.

Saisie de données

Il est possible d'affecter à une variable, une valeur entrée par l'utilisateur par l'intermédiaire du clavier :

Syntaxe : ***Entrer*** *(nomvar) ;*

X

EXERCICES D'ALGORITHMIQUE

1) Enoncés des exercices corrigés

Exercice 1

Ecrire un algorithme qui, à partir de 3 réels *a*, *b* et *c,* détermine les solutions réelles de l'équation $ax^2 + bx + c = 0$ (ou indique qu'il n'y a pas de solution).

Exercice 2

Une année est bissextile si elle est multiple de 4 sauf si c'est une année séculaire (multiple de 100). Cependant les années séculaires multiples de 400 sont bissextiles.

1) Ecrire un algorithme qui indique si une année n est bissextile, en utilisant uniquement des tests simples (pas de connecteur logique ***et*** ni ***ou***).

2) Même question en utilisant un unique test ; ***et*** et ***ou*** sont ici autorisés.

Exercice 3

On considère la suite $u_n = \sum_{i=1}^{n} \frac{1}{i}$.

1) Ecrire un algorithme qui calcule le terme de rang n.

2) Cette suite est croissante et a pour limite $+\infty$ quand n tend vers $+\infty$. Pour tout réel $A > 0$, on peut donc trouver un entier n_0 tel que : $n > n_0 \Rightarrow u_n > A$. Ecrire un algorithme qui, à partir d'un réel A, détermine le plus petit entier n_0 tel que $u_{n_0} > A$.

Exercice 4

Un entier naturel est dit premier si et seulement si il admet exactement 2 diviseurs distincts, 1 et lui-même. Par exemple 2, 3, 5, 7 sont premiers, et 0, 1, 4, 9 ne sont pas premiers.

1) Ecrire un algorithme qui indique si un entier p est premier ou non.

2) Améliorer l'algorithme en utilisant la propriété suivante : si p est non premier il existe un nombre entier d distinct de 1 tel que d divise p et $d^2 \leq p$.

3) Dénombrer les opérations pour chacun de ces algorithmes et conclure.

Exercice 5

Soient $a_1, \ldots, a_n$, n naturels non nuls. On appelle $pgcd(a_1,\ldots,a_n)$ le plus grand des diviseurs communs des a_i. Par exemple : $pgcd(6,10,14) = 2$. On montre que ce $pgcd$ existe et qu'il est unique.

1) Ecrire un algorithme qui donne le $pgcd$ de deux entiers a et b.

2) On rappelle la propriété suivante : soient 3 entiers naturels non nuls a, b et r tels qu'il existe q vérifiant $a = bq + r$, alors $pgcd(a,b) = pgcd(b,r)$.
Ecrire une nouvelle version de l'algorithme en utilisant cette propriété (algorithme d'Euclide).

X

2) Corrigés des exercices

Les solutions des exercices sont commentées. Pour les exercices 4 et 5, nous allons notamment faire « tourner à la main » les algorithmes, ce qui permettra de mieux suivre leur déroulement.

Exercice 1

```
Entrer (a) ; Entrer (b) ; Entrer (c) ;
Si a = 0 alors Si b = 0 alors Si c = 0 alors Afficher ('Tout x est solution')
                                        sinon Afficher ('Pas de solution')
                                        fin de test si ;
                              sinon Afficher ('Une solution : ', −c/b )
                              fin de test si ;
sinon d ← b*b − 4*a*c ;
      Si d > 0 alors Afficher ('Deux solutions : ', (−b − √d)/(2*a), (−b + √d)/(2*a) )
      sinon Si d = 0 alors Afficher ('Une solution : ', −b/(2*a) )
            sinon Afficher ('Pas de solution')
            fin de test si ;
      fin de test si ;
fin de test si ;
```

Commentaire : Le cas particulier $a = 0$ doit être étudié en premier lieu. Dans ce cas apparaît un « sous cas » particulier : $b = 0$. Dans ce cas apparaissent deux possibilités selon la valeur de c. Les différents tests sont donc ***imbriqués***. Lorsque a est non nul (« branche sinon » du premier test), on retrouve le calcul classique du discriminant et les tests sur son signe. Ces tests sont aussi ***imbriqués*** puisque les différents cas sont incompatibles.
Les placer en ***séquence*** comme suit :

```
Si d > 0 alors Afficher ('2 solutions : ', (−b − √d)/(2*a), (−b + √d)/(2*a) ) fin de test si ;
Si d = 0 alors Afficher ('1 solution : ', −b/(2*a) ) fin de test si ;
Si d < 0 alors Afficher ('Pas de solution') fin de test si ;
```

conduit à les réaliser systématiquement alors que si le premier est positif, les suivants sont inutiles. Lorsqu'ils sont imbriqués, seuls les tests nécessaires sont réalisés.

Exercice 2

1)

```
Entrer (n) ;
Si E( n/400 ) = n/400 alors Afficher ('Bissextile')
sinon Si E( n/100 ) = n/100 alors Afficher ('Non bissextile')
      sinon Si E( n/4 ) = n/4 alors Afficher ('Bissextile')
            sinon Afficher ('Non bissextile')
            fin de test si ;
      fin de test si ;
fin de test si ;
```

Commentaire : Les années séculaires sont des cas particuliers de multiples de 4 et les années multiples de 400 sont des cas particuliers d'années séculaires. Les tests sont donc ***imbriqués*** et on commence par les multiples de 400, puis les multiples de 100, puis les multiples de 4.

2)

```
Entrer (n) ;
Si E( n/400 ) = n/400 ou (E( n/4 ) = n/4 et E( n/100 ) ≠ n/100 ) alors
      Afficher ('Bissextile')
sinon Afficher ('Non bissextile')
fin de test si ;
```

X

COMMANDES DE BASE DE MAPLE

1) Généralités et principes élémentaires

Maple est un logiciel de calcul formel. Il effectue des calculs exacts sur des expressions littérales et n'évalue numériquement que si on le lui demande. Maple sait dériver, intégrer, effectuer des développements limités, tracer des courbes etc.
Une autre spécificité de Maple est d'effectuer les instructions, au fur et à mesure qu'on les valide. On peut ainsi observer les résultats de chaque action constituant l'algorithme complet.

Le séparateur d'instruction

Chaque instruction Maple doit être terminée par « **;** » ou « **:** ». On valide ensuite en appuyant sur la touche « ENTREE ». Si l'instruction est terminée par « **;** », l'ordre est exécuté et l'évaluation du résultat est affiché à l'écran. Si l'instruction est terminée par « **:** », l'ordre est exécuté mais rien ne s'affiche à l'écran.

On peut écrire plusieurs commandes (séparées par « **;** » ou « **:** ») sur une même ligne avant de les valider. Elles sont alors exécutées l'une après l'autre, en commençant par celle qui est la plus à gauche.

Remarque : Le point virgule peut être utilisé pour afficher la valeur d'une variable *nomvar*, en validant simplement : `> nomvar;`
Aucune action n'est réalisée, mais *nomvar* est évaluée et sa valeur est affichée.

Virgule

Pour les nombres décimaux, utilisez le caractère « **.** » à la place de « **,** » pour écrire les parties décimales.

Nom des variables

Pour nommer les variables, on choisit une combinaison de chiffres et de lettres, commençant par une lettre.

Remarque : Il est éventuellement possible d'utiliser un nom de variable contenant des blancs. Il faut alors placer ce nom entre « ` » (caractère obtenu à l'aide de la touche « Alt Gr » avec la touche « 7 », suivi d'un blanc (barre d'espace)).

Mots réservés

On ne peut pas utiliser certaines combinaisons de lettres pour définir des variables. Il s'agit de ***mots réservés*** qui correspondent à des commandes Maple, ou **Pi** qui correspond au nombre π. Certaines lettres sont aussi réservées comme **D** (qui est la commande permettant de dériver), **I** (qui représente le nombre imaginaire i).

Type des variables

Une variable possède une nature appelée ***type*** que Maple lui attribue dès que la variable est utilisée. Ce type dépend de la valeur affectée à la variable. Par exemple, si une variable A prend la valeur 1, A sera de type **integer** (entier). Si A prend la valeur $\sqrt{2}$, elle

EXEMPLES D'UTILISATION DE MAPLE

1) Enoncés des exercices

Exercice 1 — Fonctions, expressions, résolutions, graphiques.

1) Définir la fonction $g : x \to \dfrac{x^4(1-x)^4}{1+x^2}$.

2) Calculer $\int_0^1 g(x)\,dx$.

3) Déterminer l'abscisse x_m du maximum de g sur $]0,1[$ et en déduire g_{max}.

4) Tracer la représentation graphique de g sur $[0,1]$.

5) En déduire une valeur approchée rationnelle de π à un ε près que l'on précisera.

6) Vérifier le résultat précédent.

Exercice 2 — Fonctions, expressions, résolutions, graphiques.

On donne 3 points $A(x_A,y_A)$, $B(x_B,y_B)$ et $C(x_C,y_C)$.

1) Déterminer l'équation de la parabole qui passe par ces trois points.

2) Choisir des valeurs numériques et tracer le graphique faisant apparaître les 3 points et la parabole.

3) Déterminer les équations paramétrées des paraboles passant par $A(-2,1)$ et $B(1,1)$.

4) Réaliser le tracé animé des paraboles (en faisant varier le paramètre des équations) tout en faisant apparaître les points A et B.

Exercice 3 — Fonctions, expressions, résolutions, graphiques.

Un voyageur prend le train tous les jours. Chaque jour, il arrive sur le quai de la gare à l'instant où son train démarre...

Le voyageur qui se trouve initialement à une distance de $75\ m$ de la portière, court à une vitesse constante de $24\ km/h$. Le train est animé d'un mouvement rectiligne d'accélération constante $a = 1{,}2\ m/s^2$.

1) Définir la fonction x_1 (fonction du temps t en secondes) correspondant à la loi horaire du mouvement du voyageur. On prendra comme origine du repère la position du voyageur quand le train démarre.

2) Définir la fonction x_2 (fonction du temps t en secondes) correspondant à la loi horaire du mouvement de la portière du train.

3) Résoudre l'équation $x_1(t) = x_2(t)$. Qu'en déduisez-vous ?

4) Vérifier cette conclusion en représentant dans un même repère les fonctions x_1 et x_2 pour $t \in [0,30]$.

5) Définir la fonction d (fonction du temps t en secondes) donnant la distance séparant le voyageur de la portière du train. A quelle distance minimale de la portière le voyageur parviendra-t-il ?

6) Afin de ne pas manquer son train, le voyageur décide de courir plus vite, mais toujours à vitesse constante. Redéfinir la fonction x_1, puis déterminer la vitesse minimale (en m/s) à laquelle il doit courir pour attraper son train. Sur quelle distance le voyageur aura-t-il couru ?

X

2) Corrigés des exercices

Exercice 1

```
1) > g:=x->x**4*(1-x)**4/(1+x**2);
2) > int(g(x),x=0..1);
3) > solve(D(g)(x)=0,x);
```

Maple trouve plusieurs solutions (0, 1, une solution réelle et des solutions complexes). Pour récupérer la seule solution réelle dans]0,1[, il faut utiliser **fsolve** :

```
   > xm:=fsolve(D(g)(x)=0,x,avoid={x=0,x=1});
   > g(xm);
4) > plot(g,0..1);
```

5) L'intégrale calculée sur [0,1] est majorée par l'aire du rectangle de largeur 1 et de hauteur g_{max}. On en déduit donc que 22/7 est une valeur approchée de π à g_{max} près.

```
6) > evalf(22/7-Pi);
```

On verifie que cet écart est inférieur à g_{max}.

Exercice 2

```
1) > y:=x->a*x**2+b*x+c;
   > eq1:=y(xA)=yA; eq2:=y(xB)=yB; eq3:=y(xC)=yC;
   > solve({eq1,eq2,eq3},{a,b,c});
   > assign(%);
2) > xA:=1; yA:=1; xB:=2; yB:=-4; xC:=-2; yC:=0;
   > dessin1:=plot(y,-3..3):
   > dessin2:=plot([[xA,yA],[xB,yB],[xC,yC]],color=blue,symbol=box,
     style=point):
   > with(plots):display({dessin1,dessin2});
   > unassign('a,b,c');
3) > xA:=-2; yA:=1; xB:=1; yB:=1;
   > solve({eq1,eq2},{b,c});
   > assign(%);
4) > dessin1:=animate(y(x),x=-3..3,a=-1..1):
   > dessin2:=plot([[xA,yA],[xB,yB]],color=blue,symbol=box,
     style=point):
   > display({dessin1,dessin2});
```

Exercice 3

```
1) > x1:=t->24/3.6*t;
2) > x2:=t->0.6*t**2+75;
3) > solve(x1(t)=x2(t),t);
```

Maple ne trouve pas de solution réelle. Il semble que le voyageur ne pourra pas attraper le train...

```
4) > plot({x1,x2},0..30);
```

Il n'y a effectivement pas d'intersection des deux courbes.

```
5) > d:=x2-x1; solve(D(d)(t)=0,t); dmin:=d(%);
```

Le voyageur se rapprochera au minimum à 56,5 *m* de la portière.

```
6) > x1:=t->v*t;
   > solve(d(t)=0,t);
   > sol:=[%];
```

Les solutions réelles (positives) de cette équation n'existent que si la vitesse v est supérieure à sqrt(180). Pour v = sqrt(180), les deux solutions sont identiques (discriminant nul de l'équation du second degré à résoudre).

X

PROGRAMMATION DE MAPLE

Fonctions et procédures

Une ***fonction*** est un ensemble d'instructions qui calcule un ***résultat*** à partir de ***paramètres***. Ce résultat peut être utilisé, comme toute valeur, dans un calcul, une affectation, un test, une boucle ou un affichage. Une fonction ne peut donc pas avoir d'***effets de bord*** (affichage, entrée de donnée, modification de variable...) et doit se contenter de calculer son résultat.

Remarque : La plupart des instructions de Maple sont des fonctions : **diff** qui calcule la dérivée d'une expression, **simplify** qui calcule la simplification d'une expression, **plot** qui calcule les informations graphiques nécessaires à un tracé de courbe etc. On peut vérifier que ces fonctions n'ont pas d'effets de bord. Par exemple, **plot** n'affiche pas le graphique (c'est le « **;** » qui s'en charge lorsqu'on utilise l'instruction `> plot(...);`) et on peut affecter sa valeur à une variable avec l'instruction `> a:=plot(...):` (la variable *a* contient alors les informations du graphique qui peut être affiché par l'instruction `> print(a):` ou plus simplement par `> a;`).

Une ***procédure*** est un ensemble d'instructions qui effectuent des actions pouvant avoir des effets de bord. Une procédure n'a pas de résultat et ne peut donc pas être utilisée dans un calcul ou une affectation.

Remarque : Il existe des instructions de Maple qui sont des procédures : **restart** et **unassign** par exemple.

Déclaration d'une fonction

Maple permet de programmer de nouvelles fonctions.

Syntaxe : *nomfonc* := **proc**(p_1, p_2, ..., p_n)
local v_1, v_2, ..., v_m, *res* ;
$instruction_1$; ... ; $instruction_p$;
res ;
end proc ;

- *nomfonc* est le nom de la fonction.
- p_1, p_2, ..., p_n sont les ***paramètres*** de la fonction, c'est-à-dire les données qui lui permettrons de calculer son résultat.
- v_1, v_2, ..., v_m et *res* sont les ***variables locales*** de la fonction, c'est à dire qui n'existent que pendant l'exécution de la fonction. Elles peuvent avoir le même nom que des variables extérieures à la fonction (***variables globales***) sans qu'il y ait confusion et les variables globales ne sont alors pas modifiées par la fonction. Une fonction ne doit donc utiliser que des variables locales afin de s'assurer qu'elle n'a pas d'effets de bord.
- $instruction_1$, ..., $instruction_p$ sont les instructions réalisées par la fonction pour calculer son résultat.
- Le ***résultat*** de la fonction correspond à l'évaluation la ***dernière instruction*** réalisée dans la fonction. Il est donc commode (mais pas toujours obligatoire) d'utiliser une variable locale pour stocker ce résultat (notée *res* dans notre exemple) et de terminer la fonction par *res* ; (qui n'est certes pas une instruction, mais qui permettra de terminer par l'évaluation de cette variable dont la valeur sera le résultat de la fonction).

EXERCICES DE PROGRAMMATION DE MAPLE

1) Exemples de traductions d'exercices d'algorithmique

Exercice 3

```
1) > rang1:=proc(n)
   > local u,i;
   > u:=0;
   > for i from 1 to n do u:=u+1/i end do;
   > u;
   > end proc:
```

```
2) > rang2:=proc(A)
   > local u,n;
   > u:=1; n:=1;
   > while u<=A do n:=n+1; u:=u+1/n end do;
   > n;
   > end proc:
```

Commentaire : **rang1** et **rang2** sont des fonctions. Pour observer leur résultat il faut utiliser (par exemple) : `> print(rang1(12)):` ou `> rang1(12);`.

Exercice 6

```
1) > parfait1:=proc(n)
   > local s,i;
   > if n=1 then lprint(1,`n'est pas parfait`)
   > else
   >    s:=1;
   >    for i from 2 to n-1 do
   >       if irem(n,i)=0 then s:=s+i end if;
   >    end do;
   >    if s=n then lprint(n,`est parfait`)
   >    else lprint(n,`n'est pas parfait`)
   >    end if;
   > end if;
   > end proc:
```

Commentaire : **parfait1** est une procédure. Pour savoir si 6 est parfait on utilisera : `> parfait1(6):`. L'instruction **lprint** fonctionne comme **print**, mais permet d'afficher à partir du bord gauche de la page.

```
2) > parfait2:=proc(N)
   > local n,i,s;
   > for n from 2 to N do
   >   s:=1;
   >   for i from 2 to n-1 do
   >      if irem(n,i)=0 then s:=s+i end if;
   >   end do;
   >   if s=n then lprint(n,`est parfait`) end if;
```

X

2) Enoncés des exercices

Exercice 1 — Somme de deux listes.

On veut réaliser une fonction de deux paramètres L_1 et L_2 de type **list**, qui calcule la somme de ces listes, définie de la façon suivante : le $n^{ème}$ élément de la liste somme est la somme des $n^{èmes}$ éléments de chaque liste. Si une des listes a plus d'éléments que l'autre, on suppose que les éléments manquants sont nuls.
Par exemple, si $L_1 = [3,7,2]$ et $L_2 = [2,0,2,7,9]$, alors la somme des ces listes est $[5,7,4,7,9]$.

1) Ecrire une fonction **som** (de deux paramètres L_1 et L_2), qui calcule la somme des listes L_1 et L_2 en supposant que le nombre d'éléments de L_1 est supérieur ou égal à celui de L_2.
2) Ecrire une fonction **somme** (de deux paramètres L_1 et L_2), qui calcule la somme des listes L_1 et L_2 pour des longueurs de liste quelconque, en faisant appel à la fonction **som**.

Exercice 2 — Tri d'une liste.

On veut réaliser une fonction d'un paramètre L de type **list**, qui calcule la liste contenant les mêmes éléments que L, mais classés dans l'ordre croissant. On utilise pour cela l'algorithme suivant :

- Les éléments de la liste sont placés dans un tableau.
- On compare d'abord le premier élément du tableau aux suivants et on permute, si nécessaire, ce premier élément avec le plus petit des éléments suivants.
- On s'intéresse ensuite au second élément. On le compare de même aux suivants et on réalise la permutation si nécessaire. Cette méthode est réitérée jusqu'au dernier élément.
- Une fois le tableau trié, on peut construire la liste finale.

Ecrire une fonction **tri** utilisant cet algorithme.

Exercice 3 — Suite de Fibonacci.

La suite de Fibonacci est la suite (u_n) telle que : $u_0 = u_1 = 1$; $u_{n+1} = u_n + u_{n-1}$.
Ecrire une fonction d'un paramètre n entier, qui calcule le $n^{ème}$ terme de la suite :
1) sans utiliser la récursivité.
2) en utilisant la récursivité.
On pourra comparer les performances des deux versions en utilisant la fonction **time** de Maple, qui donne les temps d'exécution des fonctions ou procédures.

Exercice 4 — Courbe paramétrée et sa podaire par rapport à un point.

Soit une courbe paramétrée $M(t)$ définie, de classe C^1 et régulière sur un intervalle I. Soit A un point $M(a)$ de cette courbe. On appelle podaire de cette courbe, par rapport à A, l'ensemble, quand t parcourt I, des projections orthogonales de A sur la tangente en $M(t)$ à la courbe.
Ecrire une procédure de paramètres *x, y, a, inf* et *sup* avec x et y de type **variable fonctionnelle** (équations paramétriques de la courbe) et *a, inf* et *sup* de type **real**. La procédure tracera sur un même graphique, la courbe $M(t) = (x(t), y(t))$ et sa podaire par rapport à $M(a)$ pour t variant entre *inf* et *sup*.

3) Corrigés des exercices

Exercice 1

```
1) > som:=proc(L1,L2)
   > local L,i;
   > L:=[];
   > for i from 1 to nops(L2) do
   >    L:=[op(L),op(i,L1)+op(i,L2)]
   > end do;
   > for i from nops(L2)+1 to nops(L1) do
   >    L:=[op(L),op(i,L1)]
   > end do;
   > L;
   > end proc:
```

Commentaire : On somme les éléments des listes jusqu'au nombre d'éléments de la plus courte (L_2) et on ajoute ces sommes à la liste L. Pour terminer, il faut ajouter dans L, les éléments complémentaires de la liste la plus longue (L_1).

```
2) > somme:=proc(L1,L2)
   > local L;
   > if nops(L1)>nops(L2) then L:=som(L1,L2) else L:=som(L2,L1)
     end if;
   > L;
   > end proc:
```

Commentaire : Il suffit d'utiliser la fonction **som** en prenant soin de fournir en premier paramètre, la liste la plus longue.

Exercice 2

```
   > tri:=proc(L)
   > local T,Lf,i,j,c;
   > T:=array(1..nops(L));
   > for i from 1 to nops(L) do T[i]:=op(i,L) end do;
   > for i from 1 to nops(L)-1 do
   >    for j from i to nops(L) do
   >       if T[j]<T[i] then c:=T[i]; T[i]:=T[j]; T[j]:=c; end if;
   >    end do;
   > end do;
   > Lf:=[];
   > for i from 1 to nops(L) do Lf:=[op(Lf),T[i]] end do;
   > Lf;
   > end proc:
```

Commentaire : Le tableau doit être déclaré avant d'être utilisé.

Exercice 3

```
1) > fibo1:=proc(n)
   > local u,v,w,i;
   > u:=1; v:=1;
   > for i from 2 to n do
   >    w:=u+v; u:=v; v:=w;
   > end do;
   > v;
   > end proc:
```