

# TP Maple 3

## Listes, tests, boucles et procédures

Les structures de branchement (tests) et de répétition (boucles) sont au fondement de la programmation informatique. Elles permettent respectivement d'effectuer certaines tâches en fonction de conditions choisies par l'utilisateur et d'effectuer une tâche à plusieurs reprises.

1	Les ensembles	1
2	Les séquences	2
3	Les listes	3
4	Tests	4
4.1	Les expressions booléennes	4
4.2	La structure de test <b>if ... then</b>	5
5	Boucles	6
5.1	La boucle <b>for</b>	6
5.2	La boucle <b>while</b>	7
5.3	Exemple : calcul d'une valeur approchée	8
6	Les procédures	9
6.1	Définition d'une procédure	9
6.2	Procédures récursives	10
7	Exercices	12

### 1. Les ensembles

Un ensemble (*set* en anglais) est une énumération entre accolades d'expressions séparées par des virgules. Les éléments ne sont pas ordonnés et chaque élément ne figure qu'une seule fois. Le test booléen `member(a,E)` renvoie *true* si  $a \in E$  et *false* sinon.

```
> E:={1,5,4,5,y}; member(4,E); member(x,E);
```

$$E := \{1, 4, 5, y\}, true, false$$

Remarquons qu'un ensemble est une expression dont les opérandes sont les éléments<sup>1</sup>.

1. Attention, l'ordre des éléments d'un ensemble est imprévisible !

```
> op(E);
```

```
1, 4, 5, y
```

### Manipulation des ensembles

- ▶ **member(x,E)** : renvoie *vrai* si  $x$  appartient à  $E$ .
- ▶ **A union B** : renvoie  $A \cup B$ .
- ▶ **A intersect B** : renvoie  $A \cap B$ .
- ▶ **A minus B** : renvoie  $A \setminus B$ .
- ▶ **nops(E)** : renvoie le cardinal de  $E$ .

## 2. Les séquences

Une séquence (ou suite) sous Maple est *une énumération d'expressions séparées par des virgules*. Contrairement aux ensembles, les redondances et l'ordre des éléments sont importantes lors de la définition d'une séquence. Les séquences sont concaténables, c'est-à-dire *juxtaposables*.

```
> S1:=a,b-1; S2:=1,2,2; S3:=S1,S2,A;
```

```
S1 := a, b - 1      S2 := 1, 2, 2      S3 := a, b - 1, 1, 2, 2, A
```

On peut évaluer les séquences contenant des variables, et on peut en isoler des éléments.

```
> b:=2: S1; S3[6], S3[5];
```

```
a, 1
```

```
A, 2
```

Signalons en conclusion que les séquences ne sont accessibles qu'en seule lecture.

```
> S3[1]:=1;
Error, cannot assign to an expression sequence
```

La commande `seq` permet à l'utilisateur de réaliser des listes à partir d'une formule générale.

```
> S4:=seq(2/k,k=1..7);
```

```
S4 := 2, 1, 2/3, 1/2, 2/5, 1/3, 2/7
```

Attention ! La variable  $k$  doit être non-affectée et l'intervalle de variation de  $k$  doit être un intervalle fixé d'entiers.

On peut également employer le symbole « \$ » pour construire des séquences. Dans ce cas, les variables non-affectées sont autorisées aux bornes de variation de  $k$ .

```
> S5:=2/k $ k=1..n: eval(subs(n=7,S5));
```

$$2, 1, \frac{2}{3}, \frac{1}{2}, \frac{2}{5}, \frac{1}{3}, \frac{2}{7}$$

### 3. Les listes

Les listes sont des objets essentiels. Elles permettent notamment le traitement de séries de résultats en ingénierie. Une liste est *une énumération ordonnée d'expressions entre crochets séparées par des virgules*. Autrement dit, une liste n'est ni plus ni moins qu'une séquence entre crochets.

```
> L1:=[1,3,45,875,964];
```

$$L1 := [1, 3, 45, 875, 964]$$

```
> L2:=[seq(2*k,k=2..5)];
```

$$L2 := [4, 6, 8, 10]$$

Contrairement aux séquences, mais à l'instar des ensembles, les listes sont des expressions.

— Liste et expression —

La liste

$$[a_1, a_2, \dots, a_n]$$

est considérée par Maple comme une expression dont les opérandes sont :  $a_1, a_2, \dots, a_n$ .

Ainsi, la commande `op` retourne-t-elle la séquence formée par les éléments d'une liste.

```
> op(L2);
```

$$4, 6, 8, 10$$

On fusionne alors des listes en passant par simple concaténation des séquences correspondantes.

```
> L3:=[op(L1),op(L2)];
```

$$L3 := [1, 3, 45, 875, 964, 4, 6, 8, 10]$$

Les listes sont accessibles en lecture et en enregistrement (contrairement aux séquences qui ne sont accessibles qu'en seule lecture) : on peut modifier une liste au cours d'un programme.

```
> L2[1]:=w: L2;
```

$$[w, 6, 8, 10]$$

Les commandes `subs` et `subsop` permettent respectivement l'évaluation d'une liste et la modification de certains termes.

```
> L3:= [3, x+y, 2*x]: L3; subs(x=3, L3); subsop(3=x, L3);
```

$$[3, x + y, 2 * x]$$

$$[3, 3 + y, 6]$$

$$[3, x + y, x]$$

Voici un aperçu non exhaustif de commandes pour la manipulation des listes sous Maple...

#### — Manipulation des listes —

- ▶ `nops(L)` : renvoie le nombre d'éléments de la liste  $L$ .
- ▶ `[op(p..q), L]` : renvoie la sous-liste des éléments indexés de  $p$  à  $q$  de  $L$ .
- ▶ `[op(L1), op(L2)]` : fusionne les deux listes  $L1$  et  $L2$ .
- ▶ `subs(x=a, L)` : évalue  $L$  en  $x = a$ .
- ▶ `subsop(k=A, L)` : affecte la valeur  $A$  à  $L[k]$ .
- ▶ `select` : permet de sectionner des termes d'une liste selon un critère.
- ▶ `sort` : permet de trier une liste par ordre croissant des termes lorsque cela a un sens.

## 4. Tests

### 4.1. Les expressions booléennes

Par définition, une variable booléenne ne prend que deux valeurs : vrai (true) ou faux (false). Le logiciel manipule ce type de variables. On pourra comparer des expressions de même type à l'aide des opérateurs suivants.

#### — Opérateurs de comparaison —

- |  |  |                                       |                                   |
|--|--|---------------------------------------|-----------------------------------|
| ▶ <code>=</code> : signe = (égalité au sens mathématique). | ▶ <code>&lt;&gt;</code> : signe $\neq$ . | ▶ <code>&gt;=</code> : signe $\geq$ . | ▶ <code>&gt;</code> : signe $>$ . |
|  | ▶ <code>&lt;=</code> : signe $\leq$ .    | ▶ <code>&lt;</code> : signe $<$ .     |                                   |



```
> a:=4: if a<3 then 'a est strictement inférieur à 3' fi;
      if a>=3 then 'a est supérieur à 3' fi;
```

a est supérieur à 3

L'exemple précédent utilise deux environnements de test. On peut le programmer en un seul avec la commande `else` qui signifie « sinon ».

#### Syntaxe d'un test (II)

```
if condition then groupe 1 else groupe 2 fi;
```

où *groupe 1* est un ensemble de commandes à effectuer si la *condition* est vérifiée et *groupe 2* est un ensemble de commandes à effectuer dans le cas contraire.

```
> a:=4: if a<3 then 'a est strictement inférieur à 3' else 'a est supérieur à 3' fi;
```

a est supérieur à 3

Dans le cas où il y a plus de deux conditions, on utilisera la syntaxe indiquée ci-dessous où la commande `elif` est une contraction de « else-if ».

#### Syntaxe d'un test (III)

```
if condition 1 then groupe 1
elif condition 2 then groupe 2
elif condition 3 then groupe 3
:
:
elif condition n then groupe n
else groupe n+1 fi;
```

## 5. Boucles

Les boucles servent à effectuer un même groupe de commandes plusieurs fois à la suite.

### 5.1. La boucle for

Lorsque l'utilisateur n'a pas besoin d'effectuer de test d'arrêt lors d'une boucle, par exemple dans le cas où le nombre de passages par la boucle est connu d'avance, il peut utiliser une boucle `for`. La syntaxe sous Maple est la suivante.

#### Boucle for (I)

```
for x from x1 to x2 by p do groupe od :
```

où  $x$  est une variable non nécessairement entière,  $x_1, x_2$  et  $p$  sont des réels et *groupe* un ensemble d'instructions. Cette ligne de programmation effectue l'algorithme suivant : pour  $x$  variant de  $x_1$  à  $x_2$  avec un pas de

$p$ , exécuter les commandes *groupe*. La spécification du pas  $p$  est une option ; par défaut il vaut 1.

Le programme suivant calcule le terme d'indice 100 de la suite récurrente définie par  $u_0 = 1$  et  $\forall n \geq 0$ ,  $u_{n+1} = \ln(1 + u_n)$ .

```
> u:=1:
  for k from 1 to 100 do u:=ln(1+u): od:
  evalf(u);

0.01985723463
```

La variable de comptage ( $k$  dans l'exemple ci-dessus) n'est pas muette : elle vaut la dernière valeur calculée avant le test arrêtant la boucle.

```
> u:=1:
  for k from 1 to 100 do u:=ln(1+u): od:
  k;

101
```

Il est également possible de la laisser parcourir une liste donnée :

```
_____ Boucle for (II) _____
                               for x in L do groupe od :
```

Le programme suivant crée la séquence des 5 premiers nombres pairs.

```
> L:=[0,1,2,3,4]: S:=NULL:
  for k in L do S:=S,2*k: od:
  S ;

0, 2, 4, 6, 8
```

## 5.2. La boucle while

La boucle **while** permet d'arrêter en cours de route les itérations dès qu'une certaine condition est vérifiée — c'est ce qu'on appelle le test d'arrêt de la boucle. La syntaxe sous Maple est la suivante.

```
_____ Boucle while _____
                               while condition do groupe od ;
```

Cette ligne de programmation effectue l'algorithme suivant : tant que la *condition* est vraie, effectuer les commandes *groupe*. On prendra garde aux boucles infinies, c'est-à-dire celles qui ne se terminent jamais parce que la condition est toujours vérifiée au cours des itérations ! Dans le cas d'une « boucle folle », on tentera un

arrêt grâce à l'onglet **stop** de la barre d'outil. Le programme suivant calcule la séquence des cubes d'entiers inférieurs à 50.

```
> k:=1: S:=NULL: while k^3<=50 do S:=S,k^3: k:=k+1: od;
```

```
S := 1
```

```
k := 2
```

```
S := 1, 8
```

```
k := 3
```

```
S := 1, 8, 27
```

```
k := 4
```

Ajoutons que l'utilisateur peut empêcher l'affichage des calculs itérés de la boucle en remplaçant **od ;** par **od :**. Il faudra alors forcer le logiciel à afficher le résultat *après* la boucle.

```
> k:=1: S:=NULL: while k^3<=50 do S:=S,k^3: k:=k+1: od: S;
```

```
1, 8, 27
```

Une boucle permet par exemple de calculer la valeur d'une somme ou d'un produit. Cette structure est également adaptée au calcul des termes d'une suite récurrente. Le programme suivant calcule la somme des 100 premiers entiers naturels<sup>2</sup>.

```
> somme:=0: k:=0: while k<=99 do somme:=somme+k: k:=k+1: od: somme;
```

```
4950
```

Précisons que l'on peut combiner les deux types de boucles **for** et **while**. La syntaxe est la suivante :

\_\_\_\_\_ Boucle **for** avec test d'arrêt \_\_\_\_\_

```
for x from x1 to x2 by p while condition do groupe od :
```

### 5.3. Exemple : calcul d'une valeur approchée

Soit  $f$  la fonction définie sur  $\mathbb{R}$  par  $f(x) = x^7 + x + 1$ . Cette fonction est dérivable sur  $\mathbb{R}$  et

$$\forall x \in \mathbb{R}, f'(x) = 7x^6 + 1 > 0.$$

De plus

$$\lim_{x \rightarrow -\infty} f(x) = -\infty \text{ et } \lim_{x \rightarrow +\infty} f(x) = +\infty.$$

2. Signalons que la commande **add** effectue plus rapidement ce calcul : la ligne de commandes **add(i,i=0..99)** ; renvoie le résultat 4950.

La fonction  $f$  réalise donc une bijection strictement croissante de  $\mathbb{R}$  sur  $\mathbb{R}$ . L'équation  $f(x) = 0$  admet par conséquent une unique solution  $\alpha$ . Le programme suivant calcule une valeur approchée de  $\alpha$  à  $10^{-3}$  près par *dichotomie*<sup>3</sup>.

```
> f:=x->x^7+x+1: Digits:=4: a:=-1: b:=1:
  while b-a>0.0001 do
    if f((a+b)/2)<0 then a:=(a+b)/2:
    else b:=(a+b)/2:
    fi:
  od:
evalf(a);

-0.7966
```

## 6. Les procédures

Le programme suivant calcule le plus grand de deux nombres réels  $x$  et  $y$ .

```
> x:=1: y:=2: if x>y then Max:=x else Max:=y: fi: Max;

2
```

L'inconvénient d'un tel programme est qu'il faut modifier l'affectation de  $x$  et  $y$  avant de lancer le corps du programme... L'idéal serait utile de disposer d'une écriture semblable à celle des fonctions : définir une fonction `Max` renvoyant  $\text{Max}(x, y)$  sans avoir à écrire systématiquement le programme ou les affectations de  $x$  et  $y$  en utilisant le copier-coller. C'est précisément l'objet des procédures sous Maple.

### 6.1. Définition d'une procédure

Une procédure est un programme pouvant prendre un nombre variable d'arguments (comme une fonction) et renvoyant un résultat selon la règle suivante.

#### Règle de dernière évaluation

La valeur renvoyée par une procédure est le résultat du dernier calcul exécuté au sein du corps de programme.

La procédure suivante, prenant  $x$  et  $y$  pour arguments, calcule  $\max(x, y)$ .

```
> Max:= proc(x,y)
  if x>y then x
  else y
  fi:
end proc:
```

Il suffit alors d'appeler la procédure « `Max` » en des valeurs numériques de  $x$  et  $y$ .

3. cf. le cours d'analyse et en particulier la preuve du théorème des valeurs intermédiaires.

```
> Max(12.223,12.2231);
```

```
12.2231
```

L'usage des procédures est incontestablement *plus souple* que l'écriture de programmes sauvages !

Une procédure peut aussi ne comporter aucun argument, dans ce cas la syntaxe de l'appel est «  $f()$  » pour une procédure  $f$ .

```
> f:=proc() local k,s; s:=0; for k from 1 to 1000
do s:=s+k^2: od: end proc:
> f();
```

```
333833500
```

Dans l'exemple précédent, on a utilisé des variables locales, c'est-à-dire des variables qui ne sont accessibles qu'au sein de la procédure: il est impossible d'accéder à leurs valeurs en dehors de la procédure. Si l'utilisateur souhaite manipuler une variable en dehors de la procédure, il la déclare globalement. Le lecteur méditera l'exemple suivant qui calcule la somme  $1^2 + 2^2 + \dots + 1000^2$ .

```
> f:=proc() local k,s; s:=0; for k from 1 to 1000
do s:=s+k^2: od: end proc:
> s:=5: f(),s;
```

```
333833500,5
```

```
> g:=proc() local k; global s; s:=0: for k from 1 to 1000
do s:=s+k^2: od: end proc:
> s:=5: g(),s;
```

```
333833500,333833500
```

Syntaxe pour la création d'une procédures

```
nom := proc ( $a_1, \dots, a_n$ )
local  $\ell_1, \dots, \ell_m$  ; global  $g_1, \dots, g_r$  ;
Corps du programme
end proc ;
```

La ligne de commande  $\mathbf{nom}(a_1, \dots, a_n)$  renvoie le dernier résultat calculé par la procédure «  $\mathbf{nom}$  ». On prendra garde à ce que Maple traite les arguments  $a_k$  comme des valeurs et non comme des variables: il est donc impossible, sous peine d'un message d'erreur « *Illegal use of a formal parameter* » de modifier au sein de la procédure les arguments  $a_k$ . On utilise dans ce cas des variables locales.

## 6.2. Procédures récursives

Il est même possible de programmer une procédure qui « s'appelle lui-même » au cours de son exécution! Ce type de procédure s'appelle procédure récursive et est adapté aux objets mathématiques définis par

réurrence. On utilise la commande **RETURN** pour forcer la valeur de sortie de la procédure. Voici l'exemple d'une procédure calculant  $n!$ .

```
> fact := proc(n)
      if n=0 then RETURN(1)
      else RETURN(n*fact(n-1)) fi end proc:
> fact(30);

265252859812191058636308480000000
```

Le fonctionnement de cette procédure est simple. L'utilisateur écrit la commande **fact(30)** ; le logiciel doit alors calculer  $30 \times \text{fact}(29)$ , il « lance » donc la procédure pour la valeur 29 ; il doit alors calculer  $29 \times \text{fact}(28)$ , il « lance » donc la procédure pour la valeur 28 et ainsi de suite... Le programme s'arrête cependant grâce au test qui affecte 1 à **fact(0)**. Finalement, on a

$$\text{fact}(30) := 30 \times \text{fact}(29) := 30 \times 29 \times \text{fact}(28) := \dots := 30 \times 29 \times \dots \times \text{fact}0 := 30 \times 29 \times \dots \times 1$$

La dernière valeur calculée est donc bien  $30!$ , valeur retournée par l'appel **fact(30)**.

Il est recommandé d'utiliser l'option **remember** dans les procédures récursives, elle augmente la vitesse d'exécution du programme en conservant en mémoire la trace de ses calculs. (Mais la vitesse dépend aussi de la version de Maple ainsi que du matériel utilisé.)

```
> time(fact(20000));

0.609
> factremember := proc(n) option remember;
      if n=0 then RETURN(1)
      else RETURN(n*factremember(n-1))
      fi: end proc:
> time(factremember(20000));

0.547
```

## 7. Exercices

### Exercice 1.

Prévoir les résultats des commandes suivantes.

```
> A:={x,y}; A:={x,y,A};
> B:={a,b,B};
```

### Exercice 2.

Créer la liste  $A$  des plus petits vingts entiers positifs de deux manières différentes : au moyen de la commande `seq` puis en utilisant `$`. Créer la liste  $B$  dont les éléments sont les carrés de la liste  $A$ .

### Exercice 3.

Executer les deux programmes suivants. Pourquoi donnent-ils des résultats différents ?

```
> n:=1 : S:=0 : while S<10
do S:=S+n : n:=n+1 : od:
n; S;

> n:=1 : while sum(k,k=1..n)<10
do n:=n+1 : od:
n; sum(k,k=1..n);
```

### Exercice 4.

On pose

$$\forall n \in \mathbb{N}^*, H_n = \sum_{k=1}^n \frac{1}{k}.$$

On admet que  $H_n$  tend en croissant vers  $+\infty$  quand  $n$  tend vers  $+\infty$ . Déterminer le plus petit entier  $n$  tel que  $H_n > 7$ .

### Exercice 5.

Vous prouverez dans le cours d'analyse réelle que la suite de terme général

$$u_n = \sum_{k=0}^n \frac{1}{k!}$$

converge vers  $e$  et que  $\forall n \geq 0, |e - u_n| \leq \frac{3}{(n+1)!}$ . En déduire un programme calculant une valeur approchée de  $e$  à  $10^{-8}$  près.

### Exercice 6.

Ecrire une boucle affichant tous les nombres premiers compris entre 1 et 1000. On utilisera `isprime`.

### Exercice 7.

Déterminer les couples d'entiers naturels consécutifs  $(a, b)$  avec  $0 \leq a, b \leq 100$  tels que  $ab + 1$  soit le cube d'un entier.

### Exercice 8.

Ecrire un programme calculant par dichotomie à  $10^{-6}$  près l'unique racine sur  $[1, +\infty[$  de  $1 + 8x - x^8 = 0$ .

### Exercice 9.

Ecrire, sans utiliser la commande `solve`, une procédure `Sol` pour résoudre l'équation  $ax^2 + bx + c = 0, x \in \mathbb{R}$ , pour tout  $(a, b, c) \in \mathbb{R}^3$ .

### Exercice 10.

Pour  $(n, m) \in \mathbb{N}^2$  on note  $S(n, m)$  le nombre de surjections de  $\llbracket 1, n \rrbracket$  sur  $\llbracket 1, m \rrbracket$ .

1. Que vaut  $S(n, n)$  pour  $n \in \mathbb{N}^*$  ? Que vaut  $S(n, m)$  si  $n < m$  ?
2. Que vaut  $S(0, 0)$  ? Et  $S(n, 0)$  pour  $n \in \mathbb{N}^*$  ?
3. Montrer que pour tout  $(n, m) \in \mathbb{N}^2, S(n+1, m) = m(S(n, m) + S(n, m-1))$ .
4. Ecrire une procédure sous Maple pour la fonction  $S : \mathbb{N}^2 \rightarrow \mathbb{N}$ .

### Exercice 11.

Il est 0h lundi 1er janvier. Ecrire une procédure qui affiche le jour de la semaine et l'heure  $n$  heures plus tard. (Vous pouvez utiliser la commande `irem`.)

**Exercice 12.***La suite bête*

Dans le tableau récursif suivant chaque ligne ne dépend que de la précédente.

1  
11  
21  
1211  
111221

1. Quelle est la ligne suivante? Démontrer que le nombre  $4n$  n'apparaîtra jamais.
2. Ecrire une procédure **LaSuivante** qui à une ligne donnée associe la suivante.
3. Ecrire une procédure **SuiteBete** qui permettra de donner la  $n$ -ième ligne de la suite bête.

**Exercice 13.**

Ecrire un procédure qui donne la décomposition d'un nombre naturel en facteurs premiers. (Le but de l'exercice est de ne pas utiliser la commande `ifactor` ; en revanche vous pouvez faire appel aux équivalences avec `mod`).

**Exercice 14.**

Ecrire un procédure qui donne le quotient et le reste de la division euclidienne de  $n \in \mathbb{N}$  par  $m \in \mathbb{N}^*$ .

**Exercice 15.**

En 1742 le mathématicien allemand Christian Goldbach (1690-1764) écrivit une lettre au mathématicien suisse Leonhard Euler dans laquelle il proposait la conjecture suivante :

*Tout nombre entier pair strictement supérieur à 2 peut être écrit comme la somme de deux nombres premiers (éventuellement deux fois le même).*

1. Ecrire une procédure **Goldbach** qui à tout entier pair  $n \geq 4$  renvoie deux nombres premiers de somme  $n$  s'il en existe ou un message d'erreur dans le cas contraire.
2. Tester la conjecture de Goldbach pour des entiers entre 4 et 1000.

A ce jour la conjecture reste ouverte... En 2008 elle a été vérifiée par ordinateur pour tous les nombres pairs jusqu'à  $1,1 \times 10^{18}$ . En 2000, afin de faire de la publicité pour le livre *Uncle Petros and Goldbach's Conjecture* de Apostolos Doxiadis, un éditeur britannique offrit un prix de 1 000 000 \$ pour une preuve de la conjecture. Le prix ne pouvait être attribué qu'à la seule condition que la preuve soit soumise à la publication avant avril 2002. Il n'a jamais été réclamé.

**Exercice 16.**

Ecrire une procédure **Inverse(L)** prenant en argument une liste **L** et renvoyant la liste obtenue en balayant **L** dans le sens décroissant des indices. On proposera deux solutions : avec et sans boucle (en utilisant `seq`).

**Exercice 17.**

Ecrire une procédure **TestCroissante(L)** prenant en argument une liste de nombres réels **L** et testant si elle est croissante. On proposera deux solutions : l'une utilisant `sort`, l'autre en utilisant une boucle.

**Exercice 18.**

Ecrire un procédure **Doublons(L)** prenant en argument une liste de nombres réels **L** et testant si elle contient un terme apparaissant au moins deux fois. On proposera deux solutions : en effectuant deux boucles imbriquées *puis* en utilisant astucieusement la commande `nops`.

**Exercice 19.**

La suite de Fibonacci est définie par

$$u_0 = 1, \quad u_1 = 1, \quad \forall n \geq 0, \quad u_{n+2} = u_{n+1} + u_n.$$

Ecrire un programme renvoyant la valeur de  $u_{1000}$ .

**Exercice 20.**

Ecrire deux procédures **Suite** et **Suiter**, respectivement non récursive et récursive, prenant en argument un entier  $n$  et renvoyant le terme  $u_n$  de la suite définie par

$$\begin{aligned} u_0 &= 1, & u_1 &= 1, \\ u_2 &= 3, & u_{n+3} &= 2u_{n+2} + u_{n+1} - u_n, \quad n \in \mathbb{N}. \end{aligned}$$

## Solutions

### Solution 1.

### Solution 2.

```
> A:= [seq(k,k=0..19)];
  A:= [k $ k=0..19]; #une autre possibilité
> B:= [A[k]^2$ k=1..nops(A)];
  f:=x->x^2: B:=map(f,A); #une autre possibilité
```

### Solution 3.

Dans le premier programme le nombre  $n$  est incrémenté une fois de plus.

### Solution 4.

```
> n:=0 : H:=0 :
  while H<=7 do n:=n+1 : H:=H+1/n : od:
  n; evalf(H);
```

### Solution 5.

```
> Digits:=8: n:=0 : u:=1 :
  while evalf(3*10^8)>evalf((n+1)!)
  do n:=n+1: u:=u+1/n!: od :
  n; evalf(u);
```

### Solution 6.

```
> E:=NULL : n:=2 :
  while n<1000 do
    if isprime(n) then E:=E,n: fi:
    n:=n+1:
  od:
  E;
```

### Solution 7.

```
> E:=NULL: n:=0 :
  while n<100 do
    if type(radnormal((n*(n+1)+1)^(1/3)),integer) then E:=E,[n,n+1]: fi:
    n:=n+1:
  od:
  E;
```

### Solution 8.

On remarque que  $f(1) > 0 > f(2)$ .

```
> f:=x->1+8*x-x^8: Digits:=7: a:=1: b:=2: while 10^6*(b-a)>1
  do
    if f((a+b)/2)>0 then a:=(a+b)/2:
    else b:=(a+b)/2:
    fi:
  od: evalf(a);
> plot(f,1..2);
```

**Solution 9.**

```
> SecondDegre:=proc(a,b,c) local Delta;
Delta:=b^2 - 4*a*c;
if a=0 and b<>0 then {-c/b}
elif a=0 and b=0 and c=0 then 'Tout nombre réel est solution'
elif a=0 and b=0 and c>0 then 'Il n'y a pas de solution'
elif Delta < 0 then 'Il n'y a pas de solution réelle'
elif Delta = 0 then {-b/(2*a)}
else {(-b + sqrt(Delta))/(2*a), (-b -sqrt(Delta))/(2*a)}
fi; end proc;
> SecondDegre(1,2,1); SecondDegre(1,0,1);

{-1}, Il n'y a pas de solution réelle
```

**Solution 10.**

1. On a  $S(n, n) = n!$  puisque dans le cas  $n = m$  les surjections de  $\llbracket 1, n \rrbracket$  sur  $\llbracket 1, m \rrbracket$  sont des bijections. Si  $n < m$  alors il n'existe pas de surjection de  $\llbracket 1, n \rrbracket$  sur  $\llbracket 1, m \rrbracket$ , donc  $S(n, m) = 0$ .

2. Pour mieux cerner de quoi il s'agit il faut revenir à la version ensembliste de la notion d'application. Une application  $X \rightarrow Y$  entre deux ensembles  $X$  et  $Y$  est définie à partir de son graphe qui est un sous-ensemble  $\Gamma$  du produit  $X \times Y$  vérifiant la propriété suivante :

$$(*) \quad \forall x \in X \exists_1 y \in Y : (x, y) \in \Gamma.$$

On a  $S(0, 0) = 1$ . En effet, on est dans le cas où  $X = Y = \emptyset$ . Alors  $X \times Y = \emptyset$  et le graphe ne peut être que l'ensemble vide; on parle alors d'application vide. Comme en plus  $Y$  est vide la condition de surjectivité est trivialement satisfaite.

On a  $S(n, 0)$  pour  $n \in \mathbb{N}^*$ . En effet, on est dans le cas où  $X \neq \emptyset$  et  $Y = \emptyset$ ; alors la condition (\*) ne peut pas être validée, donc il n'existe pas d'application  $X \rightarrow Y$ .

Remarquons en passage qu'on a aussi  $S(0, m) = 0$  pour tout  $m \in \mathbb{N}^*$ , mais pour une autre raison : si  $X = \emptyset$  et  $Y \neq \emptyset$  alors il existe une application  $X \rightarrow Y$ , à savoir l'application vide, mais elle n'est pas surjective.

3. Soit  $(n, m) \in \mathbb{N}^2$ . Pour calculer  $S(n + 1, m)$  nous devons dénombrer les possibilités de construire une application surjective  $f : \llbracket 0, n \rrbracket \rightarrow \llbracket 1, m \rrbracket$ . Nous distinguons deux cas pour une telle surjection  $f$ .

►  $f^{-1}(\{f(0)\}) \neq \{0\}$ . Cela signifie que 0 n'est pas le seul élément que  $f$  envoie sur  $f(0)$ . Ainsi la restriction

$$h = f|_{\llbracket 1, n \rrbracket} : \llbracket 1, n \rrbracket \rightarrow \llbracket 1, m \rrbracket, k \mapsto f(k),$$

est surjective. Il existe  $S(n, m)$  telles surjections  $h$ , et il y a  $m$  choix possibles pour le nombre  $f(0)$ . On a alors  $mS(n, m)$  possibilités d'obtenir  $f$ .

►  $f^{-1}(\{f(0)\}) = \{0\}$ . Cela signifie que  $f$  envoie tous les éléments non-nuls sur un image différente de  $f(0)$ . Alors l'application

$$g : \llbracket 1, n \rrbracket \rightarrow \llbracket 1, m \rrbracket \setminus \{f(0)\}, k \mapsto f(k),$$

est bien définie et surjective. Il existe  $m$  choix possibles pour  $f(0)$ , et il existe  $S(n, m - 1)$  possibilités pour une surjection  $g$  comme ci-dessus. On a alors  $mS(n, m - 1)$  manières différentes d'obtenir  $f$ .

Nous avons donc montré que  $S(n + 1, m) = m(S(n, m) + S(n, m - 1))$ .

```
4. > Surj:=proc(n,m)
if n<m or m<0 then 0
elif n=0 and m=0 then 1
else m*(Surj(n-1,m-1)+Surj(n-1,m))
fi end;
> Surj(5,3);
```

**Solution 11.**

```
> Calendrier:=proc(n) local j,h,q,jour :
h:=irem(n,24,'q'): j:=irem(q,7):
if j=0 then jour:='lundi';fi;
if j=1 then jour:='mardi';fi;
if j=2 then jour:='mercredi';fi;
if j=3 then jour:='jeudi';fi;
if j=4 then jour:='vendredi';fi;
if j=5 then jour:='samedi';fi;
if j=6 then jour:='dimanche'; fi;
return(jour,h):
```

```
end proc:
> Calendrier(20368);
```

mardi, 16

**Solution 12.**

1. 312211. Explication : quand on lit la ligne précédente 111221 on entend « trois 1 deux 2 un 1 ». Si le nombre 4 n'apparaît dans la ligne  $n$  alors dans la ligne  $n - 1$  on aurait  $xxxx$ . Donc dans la ligne  $n - 2$  on aurait  $x$  fois un  $x$  et encore  $x$  fois un  $x$ . Dans la ligne  $n - 1$  il fallit donc écrire  $(2x)x$ . Contradiction.

2.

```
> LaSuiivante:=proc(Liste1) local i, Nombre, j, Liste2;
  Liste2:=[NULL];
  i:=1;
  while i<=nops(Liste1) do Nombre:=Liste1[i];
    j:=1;
    while i+j<=nops(Liste1) and Liste1[i+j]=Nombre do j:=j+1 od;
    Liste2:=[op(Liste2), j, Nombre]; i:=i+j od;
  return(Liste2);
end proc:
> LaSuiivante([1,1,1,1,5,5,1]);
```

[4, 1, 2, 5, 1, 1]

3.

```
> SuiteBete:=proc(n) option remember: local k,L:
  L:=[1]:
  for k from 1 to n-1 do L:=LaSuiivante(L) od:
  sum(L[m]*10^(nops(L)-m),m=1..nops(L))
end proc:
> SuiteBete(10);
```

13211311123113112211

**Solution 13.**

```
> decomposition:=proc(n) local x,D,k;
  x:=n; D:=NULL; k:=2;
  while x>1 do
    if x mod k=0 then D:=D,k; x:=x/k
    else k:=k+1 fi:
  od;
  D end:
> decomposition(60);
```

2, 2, 3, 5

**Solution 14.**

```
> DivEucl:=proc(n,m) local q, r;
  r:=n: q:=0:
  while m<=r
    do r:= r - m; q:= q + 1 od;
  q, r; end:
> DivEucl(13,4);
```

3, 1

**Solution 15.**

```
> Goldbach:=proc(n) local k , couple:
  couple:=NULL:
  for k from 2 to n-2 do
    while couple=NULL do
      if isprime(k) and isprime(n-k) then couple:=k,n-k
      else k:=k+1: fi:
    od:
  od:
  if couple=NULL then 'conjecture fausse' else couple fi;
end proc:
> Goldbach(89090);

3, 89087

> A:=[]: for k from 2 to 501 do A:=[op(A),Goldbach(2*k)] od: A:
if nops(A)=1000 then 'conjecture vérifiée'
else 'conjecture fausse' fi;
```

conjecture vérifiée

**Solution 16.**

Avec une boucle :

```
> Inverse:=proc(L) local K, n;
  n:=0: K:=[]:
  while n<nops(L)
  do K:=[op(K),L[nops(L)-n]]: n:=n+1: od:
  K;
end proc:
```

Sans boucle :

```
> Inverse:=proc(L) local K;
  K:=[seq(L[nops(L)+1-n],n=1..nops(L))]
end proc:
> Inverse([2,4,4,5]);
```

[5, 4, 4, 2]

**Solution 17.**

Avec une boucle :

```
> TestCroissante:=proc(L) local n:
  n:=1:
  while n<nops(L) and L[n]<=L[n+1] do n:=n+1: od:
  if n=nops(L) then print(croissante):
  else print(non-croissante):
  fi:
end proc:
```

Sans boucle :

```
> TestCroissante:=proc(L):
  if sort(L)=L then print(croissante):
  else print(non-croissante): fi:
end proc:
> TestCroissante(-5,1,7,7);
```

croissante

**Solution 18.**

► Version brutale :

```
> Doublons:=proc(L) local E:
  E:={op(L)}:
  if nops(L)>nops(E) then print(NON-INJECTIVE)
  else print(INJECTIVE) fi:
end proc:
> Doublons([1,a]);
```

INJECTIVE

► Version avec la commande sort (fonctionne seulement avec des nombres réels, pas avec [1, a] par exemple).

```
> Doublons:=proc(L) local K, n:
  K:=sort(L): n:=1:
  while n<nops(K) and K[n]<K[n+1]
  do n:=n+1: od:
  if n=nops(K) then print(INJECTIVE):
  else print(NON-INJECTIVE): fi:
end proc:
```

► Version qui fonctionne sans restriction. Il y a deux boucles : d'abord on compare le premier terme avec tous les suivants, ensuite le deuxième terme avec tous les suivants, ensuite les troisième terme avec tous les suivants, etc. Au départ un « témoin » pense que la liste est injective ; dès qu'il s'avère le contraire le témoin change d'avis et la procédure s'arrête.

```
> Doublons:=proc(L) local k, n, TEMOIN:
  n:=1: TEMOIN:=INJECTIVE:
  while n<nops(L) and TEMOIN=INJECTIVE
```

### Solution 19.

```
> Fib:=proc(n) option remember:
  if n=0 or n=1 then RETURN(1)
  else RETURN(Fib(n-2)+Fib(n-1))
  fi: end proc:
> Fib(5);
```

### Solution 20.

```
> Suiter:=proc(n) option remember:
  if n=0 or n=1 then RETURN(1)
  elif n=2 then RETURN(3)
  else RETURN(2*Suiter(n-1)+Suiter(n-2)-Suiter(n-3))
  fi:
end proc:
> Suiter(2);
```

```
do
  k:=n+1:
  while k<=nops(L) and TEMOIN=INJECTIVE
  do if L[k]<>L[n] then k:=k+1:
    else TEMOIN:=NON-INJECTIVE: fi:
  od:
  if TEMOIN=INJECTIVE then n:=n+1:
  else TEMOIN:=NON-INJECTIVE: fi:
od: TEMOIN;
end proc:
```