

# Algorithmique avec Maple

## Table des matières

<b>1</b>	<b>Algorithmes</b>	<b>2</b>
1.1	Déclarations . . . . .	2
1.2	Lecture et affichage . . . . .	3
1.3	Traitement . . . . .	3
1.4	Exemples . . . . .	3
<b>2</b>	<b>Tests</b>	<b>3</b>
2.1	Opérateurs logiques . . . . .	4
2.2	Structure de contrôle : sélection . . . . .	4
<b>3</b>	<b>Boucles</b>	<b>5</b>
3.1	Boucle <code>for</code> . . . . .	6
3.2	Boucle <code>while</code> . . . . .	6
3.3	Exercices sur les boucles . . . . .	6
3.4	Suites . . . . .	7
3.5	Listes et ensembles . . . . .	7
3.6	Opérations sur les expressions . . . . .	9
<b>4</b>	<b>Fonctions</b>	<b>9</b>
4.1	Opérations sur les fonctions . . . . .	13
4.2	Représentation des fonctions . . . . .	13
<b>5</b>	<b>Récurtivité</b>	<b>14</b>
<b>6</b>	<b>Complexité</b>	<b>15</b>
<b>7</b>	<b>Tris</b>	<b>16</b>
<b>8</b>	<b>Exercices</b>	<b>18</b>
8.1	Généralités : tests, boucles, fonctions . . . . .	18
8.2	Récurtivité . . . . .	19
8.3	Complexité . . . . .	20
8.4	Tris . . . . .	24

<b>9</b>	<b>Corrections des exercices</b>	<b>26</b>
9.1	Généralités : tests, boucles, fonctions . . . . .	26
9.2	Récurtivité . . . . .	33
9.3	Complexité . . . . .	34
9.4	Tris . . . . .	43

## 1 Algorithmes

Le mot «algorithme» a pour origine Al Khuwarizmi, un mathématicien perse du IX<sup>e</sup>siècle.

Un algorithme est un traitement de données par un nombre fini d'opérations, en un temps fini.

Les algorithmes sont écrits à l'aide d'un *langage* dont les mots constituent le *lexique*, la grammaire est la *syntaxe* et l'étude du sens est la *sémantique*.

Les données sont enregistrées, accessibles par des noms ou symboles (identificateurs).

### 1.1 Déclarations

Les données peuvent être des entiers, des flottants qui représentent des réels avec une approximation donnée, des caractères, des chaînes de caractères *etc.*

Le type de chaque donnée et variable doit être annoncé avant tout traitement.

Certains langages, comme Maple, sont faiblement ou pas typés *i.e.* ils acceptent des données non typées. Il est préférable de typer les données car certaines fonctions ont un comportement qui varie avec le type, de plus des erreurs peuvent être découvertes grâce aux erreurs de types. Par exemple si  $h$  est de type mètre, nous aurons une erreur d'exécution en machine si nous entrons  $h := 1000$  feet (une sonde spatiale a disparu à cause d'une **erreur d'unités d'accélération** : livres et newtons).

Maple possède des fonctions de test du type, mais attention :

```
whattype ( a = 2 );
           =
type ( a = 2 , boolean );
           true
```

Avec Maple une expression peut avoir plusieurs types.

## 1.2 Lecture et affichage

La lecture des données est notée avec une flèche  $\leftarrow$  ou deux points  $:=$ . Par exemple  $A \leftarrow 2, B:=3$ .

L'affichage des données s'écrit : `Afficher A`; ou simplement le nom de la donnée suivi d'un point virgule, `A`;

## 1.3 Traitement

Une addition est un exemple d'instruction dans un algorithme :

$$C \leftarrow A + B$$

Le résultat de l'addition est enregistré en mémoire et accessible, au moins momentanément, par le nom `C`.

Maple dispose des opérations arithmétiques, et de bien d'autres fonctions sur les données numériques ou non.

## 1.4 Exemples

**Exemple 1.1** *Recette des œufs au plat. Prendre une poêle, la mettre sur un feu doux, mettre une noix de beurre (ou d'huile), prendre un œuf, lorsque le beurre est fondu casser l'œuf dessus, déposer une pincée de sel sur le blanc d'œuf. Laisser cuire jusqu'à ce que le blanc soit cuit (il est... blanc).*

**Exemple 1.2** *Résolution de l'équation de degré 2. Soient  $a, b$  et  $c$  trois entiers tels que  $a \neq 0$ .*

```
a,b,c::réel
si b^2-4ac<0 alors afficher : il n'y a pas de racines réelles
ou si b^2-4ac=0 alors afficher : la racine double est -b/2a
sinon afficher : il y a deux racines réelles,
                (-b-(b^2-4ac)^(1/2))/(2a)
                (-b+(b^2-4ac)^(1/2))/(2a)
```

## 2 Tests

Les tests sont des fonctions à valeurs booléennes *false* ou *true*.

**Exemple 2.1**  $A(x) = (x > 0)$ ;  $A(2) = \text{true}$ .

## 2.1 Opérateurs logiques

Les opérateurs logiques **or** et **and** sont définis par

<b>or</b>	true	false	et	<b>and</b>	true	false
true	true	true		true	true	false
false	true	false		false	false	false

**Proposition 2.1 (Propriétés)** *or* et *and* sont commutatives

$$A \text{ or } B = B \text{ or } A$$

$$A \text{ and } B = B \text{ and } A$$

*or* et *and* sont associatives

$$(A \text{ or } B) \text{ or } C = A \text{ or } (B \text{ or } C)$$

$$(A \text{ and } B) \text{ and } C = A \text{ and } (B \text{ and } C)$$

*or* et *and* sont distributifs l'un sur l'autre

$$A \text{ or } (B \text{ and } C) = (A \text{ or } B) \text{ and } (A \text{ or } C)$$

$$A \text{ and } (B \text{ or } C) = (A \text{ and } B) \text{ or } (A \text{ and } C)$$

**Exemple 2.2** *raisin et (blanc ou noir) = (raisin et blanc) ou (raisin et noir).*

## 2.2 Structure de contrôle : sélection

La structure de *sélection if...then* est la structure

«Si la condition 1 est vraie [ou si la condition 2 est vraie ...] alors exécuter l'action 1 [ou exécuter l'action 2].»

Les crochets contiennent des opérations optionnelles. Le nombre de conditions est quelconque mais fixé.

Le *diagramme syntaxique* est

```
if condition1 then action_1
  [ elif condition_2 then action_2 ]
  .
  .
  [ else action_n ]
end_fi;
```

La dernière action (si présente) est effectuée si aucune condition n'est satisfaite (la barre verticale désigne une opération optionnelle).

*Le mot elif est la contraction de else if*

**Exemple 2.3 (Syracuse)**  $n$  est un entier positif différent de 1 :

```
if n = 0 mod 2 then n/2 else 3*n+1 fi ;
```

(conjecture : la suite de Syracuse de tout entier prend la valeur 1, puis les nombres 4, 2, 1 se répètent).

**Exemple 2.4 (Structures imbriquées)** Solution d'une équation polynomiale de degré inférieur à 2  $ax^2 + bx + c = 0$  ( $d$  est une racine carrée du discriminant) :

```
if a <> 0 then (-b-sqrt(b^2-d))/(2*a),  
              (-b+sqrt(b^2+d))/(2*a)  
  elif b <> 0 then -c/b  
    elif c <> 0 then "pas de solution"  
    else "tout nombre est solution"  
  fi ;
```

**Exercice 1** Écrire en *pseudo-code*, i.e. en français et sans référence à Maple, les algorithmes réalisant les actions suivantes :

1. Écrire un algorithme du calcul du reste de la division euclidienne de deux entiers.
2. Calcul de la valeur absolue d'un nombre.
3. Calcul du maximum de deux nombres (par comparaison, sans utiliser la fonction `max` de Maple).
4. Calcul du maximum de trois nombres.
5. Rangement de trois mots (`string`) dans l'ordre alphabétique. Indication : `mot1 := "un" : mot2 := deux;`, l'opérateur de comparaison des chaînes est `<=`.

Puis écrire les programmes en langage Maple.

### 3 Boucles

Une *boucle* est une répétition d'une même suite d'opérations.

Maple propose deux boucles, la première impose un nombre fixe d'opérations tandis que la seconde demande une condition d'entrée.

### 3.1 Boucle for

Diagramme syntaxique :

```
for <nom> [ from <expr1> ] [ by <expr2> ] to <expr3>
    | in <expr4>
do instruction (1); [ instruction (2); ... ] od;
```

**Exemple 3.1**     $S := 0;$   
  *for*  $i=1$  *to* 12 *do*  
     $S := S + i^2;$   
  *od*;

### 3.2 Boucle while

Diagramme syntaxique :

```
while condition [ and condition(2) ... or condition(k) ... ]
do instruction (1) [ instruction (2) ... ] od;
```

**Exemple 3.2**     $S := 0;$   
   $k := 1;$   
  *while*  $k \leq 12$  *do*  
     $S := S + i^2;$   
  *od*;

### 3.3 Exercices sur les boucles

**Exercice 2** Écrire des boucles qui effectuent les actions suivantes

1. Calcul de  $x^n$  ;
2. Calcul de  $n!$  ;
3. Calcul du plus petit diviseur (strictement supérieur à 1) d'un entier ;

**Exercice 3** Combien un caissier a-t-il de façons de rendre la monnaie ( $n$  €) avec des pièces de 1, 2, 5 et 10 euros ?

**Exercice 4** Écrire un programme qui calcule le développement en binaire d'un nombre écrit en décimal.

**Exercice 5** Écrire un programme qui calcule la racine carrée d'un réel  $a$  avec la suite récurrente :

$$u_{n+1} = \frac{1}{2} \left( u_n + \frac{a}{u_n} \right)$$

### 3.4 Suites

Le type *suite* est noté `exprseq`.

**Diagramme syntaxique :**

`<expr1 > [, <expr2 > , ... , <exprn > ]`

Les expressions sont séparées par des virgules.

Les types des éléments de la suite peuvent être différents :

`L := 2 , a , " b " ;`

La suite vide est notée **NULL**.

**Opérateur de répétition :** un intervalle d'entiers est noté  $a..b$

exemple

`i ^2 $i = 1 .. 5 ;`

renvoie les carrés des entiers de 1 à 5.

Autres exemples avec l'opérateur de concaténation

`a || i $i = 1 .. 3 ;`

`ai , ai , ai`

`' a || i ' $i = 1 .. 3 ;`

`a1 , a2 , a3`

L'expression de gauche est évaluée *avant* le développement. Pour obtenir l'effet souhaité, il faut *retarder* l'évaluation à l'aide des apostrophes (*back-quotes*).

**Opérateur seq :** un autre opérateur, `seq` construit les suites

`seq ( i ^2 , i = 1 .. 3 ) ;`

### 3.5 Listes et ensembles

Avec les suites nous définissons les ensembles (type `set`), ce sont des listes *non ordonnées*

**Syntaxe :** `{ exprseq }`.

L'ensemble vide est `{}`.

Nous pouvons

1. extraire un ou plusieurs éléments

```
{a,z,e,r,t,y}[2..4];
                               {t,y,e}
{a,z,e,r,t,y}[3];
                               y
```

L'ordre est aléatoire.

2. faire l'union ou l'intersection de deux ensemble avec union et intersect.

```
{a,b} intersect {b,c};
                               b
```

3. calculer un complémentaire  $A \setminus B$

```
A minus B;
```

Les listes (ordonnées, de type list) ont pour représentation : [ exprseq ].

La liste vide est [].

Nous pourrons

1. extraire un ou plusieurs éléments d'une liste

```
[a,z,e,r,t,y][3];
                               e
```

2. changer un élément d'une liste L

```
L[4] := 2;
```

3. multiplier une liste de nombres par un nombre

```
2*[1,3];
      [2,6]
```

4. additionner deux listes

```
[a,b] + [c,d];
      [c+a,d+b]
```

à condition qu'elles aient même longueur.

mais pour ajouter un élément, il faudra programmer ou utiliser des fonctions élaborées.

### 3.6 Opérations sur les expressions

Nous disposons de trois fonctions qui permettent d'accéder aux éléments qui constituent les expressions.

**op** op(<expr>) renvoie la suite des opérandes

```
op(x+2);  
      x,2
```

x+2 est de type somme (+).

*Attention*, <expresseq> n'est pas licite, la fonction op confond les éléments de la suite avec des options. Par exemple op(i,<expr>) renvoie le  $i^{\text{e}}$  opérande.

```
E := m*c^2;  
op(1,E);  
      m
```

Pour obtenir du  $i^{\text{e}}$  au  $j^{\text{e}}$  opérande : op(i..j,<expr>).

**nops** nops(<expr>) donne le nombre d'opérandes.

**subsop** subsop(i=a,<expr>) remplace le  $i^{\text{e}}$  opérande par «a».

## 4 Fonctions

Une fonction, en informatique, a la même définition qu'en mathématiques : à chaque élément d'un ensemble de définition est associé un élément d'un ensemble but.

**Syntaxe** : <nom>(<arg1> [, arg2, ...])

**Exemple 4.1** Soient a et b deux noms :

```
a := sin;  
a(Pi/6);  
b := 1;  
b(2);  
      1
```

b se comporte comme la fonction constante.

Il y a plusieurs façons de définir une fonction

**la flèche** x → <expr> où expr est une expression valide.

**Exemple 4.2**  $f := x \rightarrow x^2+1;$   
 $g := (x, y) \rightarrow \text{sqrt}(x^2+y^2);$   
 $h := t \rightarrow [t^2, t^3];$

Attention :

```
a := sin(x); f := x -> a; f(2);
      a := sin(x)
      f := x -> a
      sin(x)
```

*a n'est pas évaluée dans la flèche.*

**unapply** définit une fonction à l'aide d'une expression. Comme la flèche.  
 Syntaxe : `unapply(<expr>, L)` où *L* est une suite ou une liste de variables.

```
f := unapply(x^2, x);
      x -> x^2
g := unapply(x^2*y+y^3, [x, y]):
h := unapply(x/3+y^2,x,y):
```

**procédure** Pour Maple les procédures sont des fonctions ou non ; *i.e.* elles renvoient une valeur ou non.

Syntaxe :

```
<nom> := proc( <suite de : nom :: type> )
      local <suite de noms>;
      global <suite de noms>;
      options <suite de noms>;
      <suite d'instructions>
end;
```

*args* est la suite d'arguments, *args[i]* est le *i*<sup>e</sup> arguments, le nombre d'arguments *nargs*, est quelconque mais on ne peut appeler une fonction avec moins d'arguments que dans sa définition.

Sauf si cela est demandé explicitement, une procédure renvoie le dernier résultat évalué.

**Exemple 4.3** Dans la procédure suivante, le dernier calcul est *i<sup>n</sup>*.

```
f := proc(n :: integer)
      local i;
      for i to n do i^2 od;
```

```
end proc;
```

```
f(3);
```

9

Renvoi d'une valeur intermédiaire avec RETURN :

```
f := proc(n::integer) local i;  
    for i to n do  
        if i = 2 then RETURN(i^2) end if;  
        i^2  
    end do  
end proc
```

```
f(4);
```

4

Le programme s'arrête après l'exécution de RETURN. Le vérifier avec

```
trace(f):  
f(100);
```

Et avec une variable auxiliaire

```
f := proc(n::integer, a) local i;  
    for i to n do  
        if i = 2 then a := i^2 end if;  
        i^2 end do;  
    i^n  
end proc
```

```
f(4, 'a');
```

625

```
a;
```

4

Il est conseillé d'utiliser 'a' car une erreur se produit si la variable a a été utilisée.

Attention : la commande print ne renvoie pas un nombre, elle est utilisée à des fins d'affichage.

**Exemple 4.4** `f1 := proc(x): x^2 end :`

```

f2 := proc(x): print(x^2) end:
f1(2)^2;
                                     16

f2(2)^2;
                                     4

                                     2
                                     ()

```

Les arguments ne *doivent pas* être modifiés dans la procédure mais être copiés à l'aide de *variables locales*. Les variables utilisées doivent être déclarées; une variable non déclarée est supposée locale ou globale, c'est-à-dire avoir été définie en dehors de la procédure. Le non respect de ces règles provoque une erreur dans le meilleur des cas et des *effets de bords* (modifications non souhaitées de variables globales) dans le pire des cas.

```
f := proc(n :: integer); for i to n do i^2 od; end;
```

Warning, 'i' is implicitly declared local  
to procedure 'f'

```
f := proc(n::integer) local i;
      for i to n do i^2 end do
end proc
```

**Évaluation** : lors de l'appel d'une procédure, les arguments sont évalués de gauche à droite, puis le nom de la procédure.

En dehors de l'appel de la procédure, l'évaluation est au dernier nom *avant la procédure*; si l'on veut accéder à la procédure, il faut utiliser `eval` ou `print` par exemple :

```
f := proc(n :: integer); 2*n+1 end;
g := f;
g;
                                     f
f;
                                     f
eval(g);
proc(n :: integer); 2*n+1 end;
```

```
whattype(print(g));
                                exprseq
whattype(eval(g));
                                procedure
```

mais les types de print(f) et eval(f) sont différents.

**Fonctions inertes** Ce sont des fonctions qui ne sont pas évaluées.

**Exemple 4.5** *Le calcul peut être inutile dans l'immédiat ou bien effectué dans un contexte particulier ;*

```
Factor(x^2+1);
                                2
                                Factor(x  + 1)

value(%);
                                2
                                Factor(x  + 1)
value(% mod 5);
                                (x + 2) (x + 3)
```

## 4.1 Opérations sur les fonctions

Une expression peut être

**dérivée** une `diff(x^2,x)`; ou plusieurs fois `diff(x^2,x$2)`;

**intégrée** `int(2x,x=0..1)` (primitive `int(2x,x)`)

De même une fonction  $f$  peut être

**dérivée**  $D(f)$  (par rapport à la seconde variable  $D[2](f)$ )

**intégrée** `int(f(x),x)` ou `int(f,0..x)`

Les fonctions peuvent être composées, soient  $g \circ f$  et  $f^2$  :

```
g @ f;
f @@ 2;
```

## 4.2 Représentation des fonctions

Syntaxe : Ci-dessous,  $f$  est la fonction ou les fonctions (liste, ensemble) à représenter,  $h$  est l'intervalle horizontal et  $v$  l'intervalle vertical. Pour les options et les fonctions de deux variables, voir l'aide.

```
plot(f, h, v, options);
```

**Exemple 4.6** Quelques courbes planes.

```
plot([sin(t), cos(t)], t=-Pi..Pi); #deux courbes
plot([sin(t), cos(t), t=-Pi..Pi]); #une seule
plot(sin(t), t);
plot(sin, cos, -Pi..Pi);
plot(sin);
plot([cos(2*x), sin(3*x), x=-Pi..Pi], -4..4, -5..5);
plot([sin(4*x), x, x=0..2*Pi], coords=polar);
plot(exp, -infinity..infinity);
```

## 5 Récursivité

Une définition est *récursive* si elle fait appel à une autre fonction, y compris à elle-même.

Une définition récursive est *divergente* si le nombre d'appel à elle-même (auto-référence) est infini. Le *cercle vicieux* apparait si la définition ne fait appel qu'à elle-même.

**Exemple 5.1 (Philippe Martin. Informatique : méthodes et applications)** –

*Une chaîne c'est un maillon ou un maillon lié à une chaîne.*

- L'ascendant généalogique d'un individu c'est son père, sa mère, ou un de leurs ascendants;
- Soit  $x$  un réel non nul, et  $n$  un entier. Alors  $x^n$  est égal à 1 si  $n = 0$  et à  $x \cdot x^{n-1}$  sinon.

**Définition 5.1** Un programme récursif est dit avoir une récursivité terminale si et seulement si le programme se termine après un appel à lui même.

**Exemple 5.2** La fonction

```
fact := proc(n :: integer);
  if n <= 1 then 1 else n*fact(n-1) fi;
end;
```

n'est pas récursive terminale. En revanche la fonction `fact_t` l'est :

```
fact_t := proc(n, f :: integer);
  if n < 1 then f else fact_t(n-1, n*f) fi;
```

*end* :

*fact\_t*(*n*, 1);

Comparer les fonctions avec *trace* et utiliser *time*() .

## 6 Complexité

La complexité en mathématiques (et informatique) est l'estimation du nombre de calculs et de la place nécessaire à ces calculs.

Si les calculs sont séquentiels, notons  $c(K)$  le coût en temps ou en place d'un calcul  $K$ , pour  $n$  calculs numérotés de 1 à  $n$ , le coût est :

$$\sum_{1 \leq j \leq n} c(K_j)$$

(unités non précisées) la fonction de coût,  $c$ , est supposée additive.

Par exemple, pour une boucle  $B$  de coût  $c(B)$  :

$$m \cdot c(B)$$

si la boucle est effectuée  $m$  fois.

Coût d'un programme récursif  $P$  faisant appel à une suite de programmes  $(P_j)_{0 \leq j \leq m}$  :

$$c(P) = \sum_{0 \leq j \leq m} c(P_j)$$

certaines programmes appelés peuvent être aussi récursifs.

**Exemple 6.1** Coût de

```
fact := proc(n :: integer)  
  local k, f;  
  k := 1;  
  f := 1;  
  while k <= n  
    do f := f * k; k := k + 1 od;  
  f;  
end;
```

La boucle est effectuée  $n$  fois, elle contient une multiplication et une addition (les affectations et le test sont négligés), le coût  $c$  est donc proportionnel à  $n$ . Lorsque la fonction de coût est majorée par une fonction affine de  $n$ , nous écrivons :

$$c(n) = O(n)$$

Version récursive :

```

fact_rec := proc (n :: integer):
  if n <= 1 then 1 else n * fact_rec (n - 1); fi;
end:

```

Le coût  $c(\text{fact})$  vérifie :

$$c(\text{fact\_rec})(n) = c(\text{fact\_rec})(n - 1) + 1$$

car l'appel de  $\text{fact}(n)$  demande les calculs de  $\text{fact}(n)$  plus une multiplication. La fonction de coût satisfait une équation de récurrence linéaire d'ordre 1 (suite arithmétique). Le coût est encore linéaire.

## 7 Tris

Les tris dont il est question ici, consistent à permuter les éléments d'une suite de manière à obtenir une suite croissante relativement à un ordre donné.

L'opération de tri se fait en déplaçant un élément suivant une règle qui est itérée.

**Tri sélection** On cherche le plus petit élément (ou le plus grand) de la liste et on le met en première position. Puis par récurrence, on cherche le  $k^{\text{e}}$  plus petit élément que l'on place en  $k^{\text{e}}$  position (le plus petit élément parmi les  $n - k + 1$  restants,  $n$  étant le nombre d'éléments de la liste).

**Tri insertion** On débute avec la liste 1 qui est la liste à trier et la liste 2 qui est vide et que l'on va remplir au fur et à mesure que l'on vide la première. On choisit un élément de la liste 1 pour démarrer la liste 2. Puis, par récurrence, on choisit un élément de la liste 1 que l'on insère à la bonne place dans la liste 2. Lorsque la liste 1 est vide, la liste 2 est la liste triée cherchée.

**Tri bulle** Le principe du tri bulle est de comparer deux éléments consécutifs et les échanger s'ils ne sont pas dans le bon ordre, en commençant par un bout de la liste et en itérant jusqu'à la fin. Après un premier tour un élément est à sa place définitive. On recommence jusqu'à ce que tous les éléments soient à leur place.

**Tri fusion** Le principe est de couper la liste en deux, trier les deux listes puis les fusionner. Donc, récursivement, on découpe une liste en deux listes, puis ces listes en deux et ainsi de suite jusqu'à ce que les listes obtenues n'aient qu'un élément au plus, elles sont alors triées, il reste à fusionner ces listes triées.

**Tri rapide** Un exemple de tri récursif par dichotomie : *le tri rapide* de C.A.R. Hoare.

Choisissons un élément appelé le pivot et divisons la liste en deux sous-listes, la première contient les éléments plus petits que le pivot et la seconde contient les autres. Nous obtenons une liste avec la première sous-liste à laquelle nous ajoutons le pivot puis la seconde sous-liste. Le pivot a donc atteint sa place définitive. Il suffit de recommencer la même opération avec les deux listes. Ce tri se programme donc facilement de manière récursive.

Nous avons besoin d'une fonction qui découpe une liste  $\ell$  en deux listes selon une propriété  $P$ . Les éléments de  $\ell$  qui vérifient  $P$  sont placés dans la liste  $\ell_1$  et les autres dans la liste  $\ell_2$ .

Une version simple de tri rapide est la suivante :

choisissons un *pivot*, par exemple le dernier élément de la liste  $L$ , puis parcourons la liste en partant de la gauche jusqu'à trouver un élément  $L[i]$  plus grand que le pivot, continuons le parcours de la liste partant de la droite jusqu'à trouver un élément  $L[j]$  plus petit que le pivot. Échangeons ces deux éléments :  $L[i] \leftrightarrow L[j]$  et continuons le balayage. Quand les balais se croisent ( $i \geq j$ ), ils définissent deux listes, mettons le pivot au milieu à la place de  $L[i]$  ( $L[i]$  est plus grand que le pivot), c'est sa place définitive.

**Exemple** Le pivot est en boîte les éléments à échanger en **gras**.

	18	20	57	61	11	48	24
i=3, j=5	18	20	<b>57</b>	61	<b>11</b>	48	24
échange	18	20	11	61	57	48	24
fin du balayage	18	20	11	<b>61</b>	57	48	24
24 est placé	18	20	11	24	57	48	61

Il reste à appliquer les mêmes opérations sur les sous-listes [18,20,11] et [57,48,61].

## 8 Exercices

### 8.1 Généralités : tests, boucles, fonctions

**Exercice 1** Écrire un algorithme d'échange du contenu de deux bols. *Indication : utiliser un troisième bol.* Application : écrire le programme Maple qui commence par  $a:=0$ :  $b:=1$ : et se termine par la commande  $a,b$ ; qui affiche 1, 0.

**Exercice 2** Écrire en *pseudo-code*, *i.e.* en français et sans référence à Maple, les algorithmes réalisant les actions suivantes :

1. Calcul de la valeur absolue d'un nombre.
2. Calcul du maximum de deux nombres (par comparaison, sans utiliser la fonction `max` de Maple).
3. Calcul du maximum de trois nombres.
4. Rangement de trois mots (`string`) dans l'ordre alphabétique. Indication :  $\text{mot1} := \text{"un"}$  :  $\text{mot2} := \text{"deux"}$ ; l'opérateur de comparaison des chaînes est  $<=$ .

Puis écrire les programmes en langage Maple.

**Exercice 3** Écrire des boucles qui effectuent les actions suivantes

1. Calcul de  $x^n$  ;
2. Calcul de  $n!$  ;
3. Écrire un algorithme du calcul du reste de la division euclidienne de deux entiers.
4. Calcul du plus petit diviseur (strictement supérieur à 1) d'un entier ;

**Exercice 4** Combien un caissier a-t-il de façons de rendre la monnaie ( $n$  €) avec des pièces de 1, 2, 5 et 10 euros ?

**Exercice 5** Écrire un programme qui calcule le développement en binaire d'un nombre écrit en décimal.

**Exercice 6 (Listes)** Écrire des algorithmes qui

1. ajoute un élément en fin de liste ;
2. insère un élément après le  $n^{\text{e}}$  élément d'une liste ;
3. indique si 0 est présent dans une liste de nombres ;

4. retire tous les 0 (on pourra utiliser la fonction *member*);
5. retire les doublons;
6. place le plus grand élément en dernière position;

En général, il y a deux possibilités : la liste d'origine est modifiée (moins de place en mémoire) ou bien une seconde liste est construite (souvent plus simple).

**Exercice 7** Écrire des fonctions qui calculent  $x^n$  et  $n!$ .

**Exercice 8** Écrire une fonction qui calcule la racine carrée d'un réel  $a$  avec la suite récurrente :

$$u_{n+1} = \frac{1}{2} \left( u_n + \frac{a}{u_n} \right)$$

(supposée convergente) à  $10^{-8}$  près.

**Exercice 9** Écrire une fonction qui calcule le développement en binaire d'un nombre entier écrit en décimal.

**Exercice 10** Écrire une fonction de calcul des coefficients binomiaux grâce à la formule du triangle de Pascal en utilisant un tableau.

**Exercice 11** Suite de Syracuse.

**Exercice 12** Le programme suivant peut être utilisé pour l'exponentiation rapide. Écrire un programme qui prend  $n$  un nombre entier en base 10 et renvoie le nombre  $n$  en base 2 :

$$\sum_{k \geq 0} a_k 10^k \mapsto \sum_{j \geq 0} b_j 2^j$$

avec, pour tout  $j, k$ ,  $a_k \in \llbracket 0, 9 \rrbracket$  et  $b_j \in \{0, 1\}$ .

**Exercice 13** Calculs des suites de Fibonacci.

## 8.2 Récursivité

**Exercice 14** Écrire et tester des programmes récursifs pour

1. le calcul de la factorielle d'un entier positif;
2. le calcul des termes de la suite de Fibonacci :

$$u_0 = 1, \quad u_1 = 1, \quad \forall n > 1 : u_n = u_{n-1} + u_{n-2}$$

3. le calcul d'une solution des équations de la forme  $f(x) = 0$  où  $f$  est continue sur un intervalle  $[a, b]$ .

### 8.3 Complexité

*Avertissement : pour les calculs de complexité en temps, nous supposons, pour simplifier, que le temps de calcul d'une addition et d'une multiplication est d'une unité. Nous ne ferons pas de calcul de complexité en place mémoire.*

**Exercice 15** Écrire un programme qui calcule les coefficients binomiaux et calculer sa complexité.

**Exercice 16 (Suites de Fibonacci)** Soit  $(F_n)_{n \geq 0}$  la suite définie par récurrence par  $F_0 = 1, F_1 = 1$  et pour  $n \geq 2 : F_n = F_{n-1} + F_{n-2}$ .

1. Écrire deux programmes, en langage Maple, de calcul de  $F_n$ , le premier avec une boucle `for`, le second sera récursif.
2. Calculer, en fonction de l'entier  $n$ , le terme général de la suite  $(c(n))_{n \geq 0}$  définie par  $c(0) = 0, c(1) = 0$  et pour  $n \geq 2$  par

$$c(n) = c(n-1) + c(n-2) + 1$$

*Il s'agit d'une équation linéaire.*

3. Calculer le nombre d'opérations (additions et multiplications) nécessaires au calcul de  $F_n$  pour chacun des programmes écrits en réponse à la première question.
4. Évaluer la complexité (comportement asymptotique du coût en nombre d'opérations de calculs) de la fonction suivante :

```
f := proc(x :: integer, y :: integer, n :: integer):  
  if n <= 1 then x else  
    f(x+y, x, n-1)  
  fi :  
end:
```

Que représente  $f(1, 1, n)$  ?

**Exercice 17 (Puissance rapide)** 1. Écrire un programme de calcul de  $x$  à la puissance entière positive  $n$ , n'utilisant que la multiplication ou des tests. Les tests prennent peu de temps et nous les négligerons dans les calculs de complexité.

2. Écrire ensuite en langage Maple l'algorithme (1) page 21. Calculer sa complexité en temps.

**Exercice 18** Il existe une relation entre la fonction binaire et la fonction `puiss_rapide`, qui est illustrée dans la fonction `puiss_bin` :

---

**Algorithme 1** (version récursive)

---

Calcul de  $x^n$ **si**  $n = 0$  **alors**

1

**si**  $n = 1$  **alors** $x$ **si**  $n$  est pair **alors** $\left(x^{\frac{n}{2}}\right)^2$ **sinon** $x * \left(x^{\frac{n-1}{2}}\right)^2$ **fin si****fin si****fin si**

---

```
puiss_bin := proc(x, n :: integer)
  local L, p, k:
  L := [binaire(n)]:
  p := x:
  for k to nops(L) do
    if L[k]=1 then p := x * p else p := p**2 fi:
  od:
end:
```

En effet le calcul de  $x^n$  en utilisant l'algorithme de la puissance rapide dépend du développement binaire de  $n$ . Si  $n = \sum_{0 \leq j \leq m} b_j 2^j$  :

$$x^n = x^{(2 \cdots 2(2b_m + b_{m-1}) + \cdots + b_0)}$$

L'algorithme est

Par exemple

$$\begin{aligned} x^7 &= x^{2^2+2+1} \\ &= x^{2(2+1)+1} \end{aligned}$$

On effectue  $x^2$  puis  $x^2 * x$ ,  $(x^2 * x)^2$  et enfin  $(x^2 * x)^2 * x$ .

Écrire des programmes Maple qui calculent  $x^n$  avec cette méthode.

**Exercice 19 (Complexité du calcul des suites de Fibonacci)** Considérons la

---

**Algorithme 2** puissance rapide et développement binaire

---

```
p ← x
pour j de m à 0 faire
  si bj = 1 alors
    p ← p * x
  sinon
    p ← p2
  fin si
fin pour
```

---

suite  $u$  définie par

$$\begin{cases} u_0 & = & 0 \\ u_1 & = & 1 \\ u_{n+2} & = & u_{n+1} + u_n \quad \text{si } n \geq 0 \end{cases}$$

C'est une suite de Fibonacci.

**Question 1** Écrire un programme de calcul des termes de la suite  $u$ , puis calculer sa complexité.

Nous avons vu des programmes dont la complexité était un  $O(n^2)$  puis  $O(n)$ . Nous allons voir ci-dessous que l'on peut obtenir mieux.

**Question 2** Démontrer que pour tous  $p$  et  $q$  tels que  $p \geq 0$  et  $q \geq 1$  :

$$u_{p+q} = u_{p+1}u_q + u_p u_{q-1}$$

**Question 3** Démontrer que  $u_{2m} = 2u_{m+1}u_m - u_m^2$ . Exprimer, de même,  $u_{2m+1}$  et  $u_{2m+2}$  en fonction de  $u_m$  et  $u_{m+1}$ .

**Question 4** Tout entier positif  $n$  s'écrit sous la forme  $n = \sum_{k=0}^{k=q} 2^k a_k$ , avec des coefficients  $a_k$  égaux à 0 ou à 1. Écrivez un programme qui, à  $n$  renvoie la liste  $[a_q, \dots, a_0]$ .

**Question 5** Soit  $R$  un algorithme récursif tels que pour tout entier  $n$ , écrit en binaire  $n = \sum_{k=0}^{k=q} 2^k a_k$ , et toute liste  $L = [a_q, \dots, a_0]$ ,  $R(m, [a_q, \dots, a_0])$  appelle  $R(2m + a_q, [a_{q-1}, \dots, a_0])$  et renvoie  $m$  si la liste  $L$  est vide.

Que renvoie  $R(0, [1, 0, 1])$  ? Et dans le cas général, que renvoie  $R(0, [a_q, \dots, a_0])$  ?

**Question 6** Démontrer que le programme récursif suivant renvoie les termes  $u_n, u_{n+1}$  lorsque l'on appelle `fib_log(0,1,b)` où  $b$  est la liste des coefficients binaires de  $n : n = \sum_{k=0}^{q-1} 2^k a_k$ .

```
fib_log := proc(u, v, b);
  if b=[ ] then u, v elif b[1] = 0 then
    fib_log(2*u*v-u^2, u^2+v^2, b[2..nops(b)]);
  else
    fib_log(u^2+v^2, 2*u*v+v^2, b[2..nops(b)]);
  fi;
end;
```

avec `b[1]` pour  $a_0$  et `b[2..nops(b)]` pour  $[a_1, \dots, a_q]$ .

**Question 7** Calculer le nombre d'appels récursifs lors de l'appel `fib_log(0,1,n)`. On pourra supposer, pour simplifier, que  $n$  est une puissance de 2 :  $n = 2^k$ .

## 8.4 Tris

**Exercice 20 (Tris)** *Pour simplifier, les tris pourrons se faire sur des listes d'entiers.*

Vous choisirez un tri ci-dessous et écrirez son algorithme.

**Question 1 (Tri sélection)**

**Question 2 (Tri insertion)**

**Question 3 (Tri bulle)**

**Question 4 (Tri fusion)**

**Question 5 (Tri rapide)** Écrire un programme qui, suivant un test scinde une liste en deux listes : la première contient les éléments qui satisfont le test et la seconde contient les éléments qui ne la satisfont pas. Par exemple les éléments plus petits qu'un entier donné. Vous appellerez ce programme `partition(L,i,j)` où `L` est une liste, `i`, l'indice du premier élément et `j` l'indice du dernier élément de la liste à traiter, car il est nécessaire de pouvoir effectuer des opérations sur des sous-listes. `partition` doit retourner la liste modifiée et l'indice `k` de l'élément qui a obtenu sa place définitive.

Exemple avec le pivot égal à 5 :

```
partition( [4,8,3,4,6,5,8,7,2,9], 5);  
          [4,3,4,5], [8,6,8,7,9]
```

Écrire, en langage Maple, le programme de tri rapide dont voici l'algorithme :

```
début tri_rapide (L::liste,i::entier,j::entier):  
  si i<j alors faire  
    partition(L,i,j):  
    tri_rapide(L,i,k-1):  
    tri_rapide(L,k+1,j):  
  fin si:  
fin tri_rapide:
```

`tri_rapide(L,i,j)`; trie la sous-liste  $L[i], \dots, L[j]$  de  $L[1], \dots, L[n]$ . Pour trier une liste, on appellera :

```
tri_rapide(L,1,n);
```

où  $n$  est l'indice du dernier élément.

**Exercice 21** Étudier la complexité des tris. Pour simplifier, dans le pire des cas, *i.e.* lorsque le nombre de calculs est maximal. Vous rechercherez également les cas dans lesquels le nombre de calculs est minimal.

## 9 Corrections des exercices

### 9.1 Généralités : tests, boucles, fonctions

**Exercice 1 (Échange)** Deux variables  $a$  et  $b$  contiennent des données.

```
variable locale e
e := a;
a := b;
b := e;
fin;
```

**Exercice 2** 1. **si**  $a \geq 0$  **alors**

```
    a
sinon
    -a
fin si
if  $a \geq 0$  then a else -a end if;
```

2. **si**  $a \geq b$  **alors**

```
    a
sinon
    b
fin si
if  $a \geq b$  then a else b end if;
```

3. Si  $a$  n'est pas le plus grand,  $b$  ou  $c$  est le plus grand. Si ce n'est pas  $c$  alors c'est  $b$ .

```
si  $a$  est plus grand que  $b$  et  $c$  alors
    a
sinon
    si  $c$  est plus grand que  $b$  alors
        c
    sinon
        b
    fin si
fin si
```

```
maks := proc(a, b, c)
    if  $b \leq a$  and  $c \leq a$  then a elif  $b < c$  then c else b end if
end proc
```

4.  $a, b$  et  $c$  sont des variables de type numérique ou chaîne.

```
if a>b then
    if c>a then c,a,b
    elif b>c then a,b,c
    else a,c,b
fi ;
elif a>c then b,a,c
elif b>c then c,b,a
else b,c,a
fi ;
```

Le programme précédent peut être raccourci en écriture (sinon en temps d'exécution) avec l'emploi d'un programme d'échange.

Nous comparons deux éléments puis nous insérons le troisième. Il s'agit d'un *tri*.

```
if a>=b then u,v:=a,b else u,v:=b,a fi ;
if c>=u then c,u,v elif v>=c then u,v,c else u,c,v fi ;
```

**Exercice 3** 1. Variables :  $m$  est l'exposant et  $p$  est égal à  $x^m$ .

```
« Initialisation »
m ← 0
p ← 1
pour m = 1 to m = n faire
    p ← p * x
fin pour
```

Avec une boucle *tant que* :

```
« Initialisation »
m ← 0 ; p ← 1
tant que m < n faire
    p ← p * x
    m ← m + 1
fin tant que
p
```

En sortie de boucle,  $m$  est égal au nombre de passages dans la boucle. Donc en entrée de boucle  $p = x^m$  et en sortie de boucle  $p = x^{m+1}$ . La sortie a lieu pour  $m = n$ .

Programmes Maple En entrée de boucle  $p$  vaut  $x^k$  et  $x^{k+1}$  en sortie.

```

k := 0;
p := 1;
for k from 1 to n
    do p := p * x; od;
p;

```

Si  $n = 0$  la boucle n'est pas exécutée.

Preuve : l'hypothèse est  $p = x^k$  en entrée de boucle. Alors  $p = x^{k+1}$  en fin de boucle. Si  $k = 0$  alors  $p = x^0 = 1$ , ce qui est vrai. La condition de sortie est  $k = n$ , d'où  $p = x^n$ .

2. Cet algorithme est semblable au précédent.

```

« Initialisation »
m ← 2; p ← 1
tant que m ≤ n faire
    p ← p * m
    m ← m + 1
fin tant que
p

```

Si  $m := 2$  alors  $p := 1$ . Supposons le compteur égal à  $m$  et  $p := (m-1)!$  en entrée de boucle. En sortie  $p := (m-1)!m$  et le compteur vaut  $m+1$ . L'arrêt se produit pour  $m = n+1$  soit lorsque  $p = n!$ .

Programme Maple :

```

k := 1;
f := 1;
while k ≤ n
    do f := f * k; k := k + 1 od;
f;

```

La preuve est semblable au cas  $x^n$ .

3. L'algorithme proposé est celui qui teste tous les entiers à partir de 2.

Division de  $a \geq 0$  par  $b > 0$  :

```

r := a;
q := 0;
while r ≥ b
    do r := r - b; q := q + 1 od;
q, r;

```

4. Utilisation possible des fonctions floor ou type(<expr>, integer).

```

d := 2;
while n/d <> floor(n/d)
  do d := d+1 od;
d;

```

**Exercice 4 (caissier)** Toutes les combinaisons sont testées.

```

m := 0;
for a from 0 to n do
  for b from 0 to n/2 do
    for c from 0 to n/5 do
      for d from 0 to n/10 do
        if a+2*b+5*c+10*d= n then m := m + 1 fi;
      od;
    od;
  od;
od;

```

**Exercice 5 (Développement binaire)** Utilisation de la fonction type : soit  $n$  un entier positif.

```

k := n: L := NULL:
while k >0 do
  if type(k, even) then L := 0, L: k := k/2
  else L := 1, L: k := (k-1)/2 fi;
od;
L;

```

$n = \sum_{0 \leq j \leq d} a_j 2^j$  avec pour tout  $j$ ,  $a_j$  égal à 0 ou 1; nous calculons dans l'ordre :  $a_0, \dots, a_d$ , le résultat affiché est  $a_d, \dots, a_0$ .

**Exercice 6 (Listes) Programmes** (langage Maple)

1. Soit  $L$  la liste et  $e$  l'élément :

```

[seq(L[k], k=1..nops(L)), e];
[op(L), e];

```

Attention  $L[k]$  est un élément et  $L[j..k]$  est une liste (même si  $j = k$ ).

2. (mêmes notations)

```

[seq(L[k], k=1..n), e, seq(L[k], k=n+1..nops(L))];
[op(1..n, L), e, op(n+1..nops(L), L)];

```

3. Renvoie true si 0 est présent et false sinon.

```
yest := false;  
k := 1;  
while (not yest) and (k<= nops(L)) do  
    yest = (L[k]=0); k := k+1;  
od;  
yest;
```

ou

```
{0} intersect {op(L)};  
member(0,L);
```

4. S est une suite auxiliaire.

```
S := NULL;  
for k from 1 to nops(L) do  
    if L[k]<>0 then S := S,L[k] fi;  
od;  
[S];
```

Plus court :

```
subs(0=NULL,L);
```

5. Utilisation de la fonction member (plutôt qu'un programme ci-dessus)

```
E := {};  
S := NULL;  
for k from 1 to nops(L) do  
    if (not member(L[k], E)) then  
        S := S, L[k];  
        E := E union {L[k]}  
    fi;  
od;  
[S];
```

où E est l'ensemble des éléments rencontrés et S la nouvelle liste. Mais E n'est pas indispensable :

```
S := NULL;  
for k from 1 to nops(L) do  
    if (not member(L[k], [S])) then  
        S := S, L[k];  
    fi;  
od;
```

```

    fi;
  od;
[S];

```

6.  $M$  est le plus grand élément.

```

M := L[1];
for k from 2 to nops(L) do
  if L[k]>M then
    T := M;
    M := L[k];
    L[k] := T;
  fi;
od;
L;

```

Mais l'ordre est changé. Pour un tri c'est acceptable. Pour garder l'ordre :

```

position := 1;
for k from 2 to nops(L) do
  if L[k] > L[position] then position := k fi;
od;
[op(1..position-1,L),op(position+1..nops(L),L),L[position]];

```

**Exercice 7 (Puissance et factorielle)** `puiss := proc(x :: real, n :: integer)`

```

  local k, p;
  p := 1;
  for k from 1 to n
    do p := p * x; od;
end;

```

```

fact := proc(n :: integer)
  local k, f;
  k := 1;
  f := 1;
  while k <= n
    do f := f * k; k := k + 1 od;
  f;
end;

```

**Exercice 8 (Racine carrée)**  $c$  est le premier terme,  $c \neq 0$ . Nous savons que la suite converge. Nous arrêtons la boucle lorsque deux termes consécutifs sont distants de moins de  $10^{-8}$

```
rc := proc(x :: real)
  local epsilon, u, v;
  epsilon := 10(-8);
  u := x;
  v := u + epsilon;
  while abs(u-v) >= epsilon and u<>0 do
    v := u;
    u := (u+v)/2;
  od;
  u;
end;
```

**Exercice 9 (Développement binaire)** bin := proc(n :: integer)

```
  local k, L;
  k := n: L := NULL;
  while k > 0 do
    if type(k, even) then L := 0, L: k := k/2
      else L := 1, L: k := (k-1)/2 fi;
  od;
  L;
end;
```

**Exercice 10 (Coefficients de Pascal)** coef\_bin := proc(m :: integer)

```
  local C, i, j;
  option remember;
  C := array(0..m, 0..m);
  C[0,0] := 1;
  C[1,0] := 1;
  C[1,1] := 1;
  for i from 2 to m do
    C[i,0] := 1;
    C[i,i] := 1;
    for j from 1 to i-1 do
      C[i,j] := C[i-1,j-1] + C[i-1,j];
    od;
  od;
```

```

    od;
    C;
end:

```

Les coefficients  $C[i, j]$  ne sont pas définis pour  $i < j$  mais nous pourrions leur affecter la valeur 0.

**Exercice 11 (Syracuse)** `syracuse := proc(n::integer)`

```

    local s;
    s:=n;
    while s>1 do
        if s mod 2 = 0 then
            s:= s/2 else s:= 3*s+1
        fi; lprint(s)
    od;
    s;
end:

```

$s/2$  peut être remplacé par `iquo(s,2)` (quotient entier) et le programme peut être accéléré en écrivant `s := (3*s+1)/2` remarquant que  $3*s+1$  est pair.

**Exercice 12**

**Exercice 13 (Suites de Fibonacci)** `fib := proc(n :: integer)`

```

    local i, u, v, w;
    u := 1;
    v := 1;
    for i from 2 to n do
        w := u;
        u := v;
        v := w + v;
    od;
    v;
end:

```

## 9.2 Récursivité

**Exercice 14** `fact:=proc(n::integer):`

```

    if n<=1 then 1 else n*fact(n-1); fi;
end:

```

Pour la procédure suivante, l'option remember améliore très sensiblement les performances.

```
fib := proc(n :: integer)
  option remember:
  if n < 2 then 1 else fib(n-1) + fib(n-2); fi;
end:
```

Ci-dessous, epsilon est la précision voulue.

```
dich := proc(f :: name, a, b, epsilon :: realcons)
  if evalf(f(a)*f(b)) <= 0 then
    if abs(a-b) < epsilon then evalf((a+b)/2) else
      if evalf(f(a)*f((a+b)/2)) <= 0 then
        dich(f, a, (a+b)/2, epsilon) else
        dich(f, (a+b)/2, b, epsilon)
      fi;
    fi;
  else print('erreur!');
  fi;
end:
```

### 9.3 Complexité

**Exercice 15** Complexité du calcul par le programme binome qui utilise la formule  $\binom{n+1}{p+1} = \binom{n}{p} + \binom{n}{p+1}$ . Soit  $t(n, p)$  le coût de binome( $n, p$ ). On a  $t(n, 0) = 0$  et, avec un coût égal à 1 pour l'addition :

$$t(n, p) = t(n, p-1) + t(n-1, p-1) + 1$$

Posons  $t'(n, p) = t(n, p) + 1$ , alors  $t'(n, p)$  vérifie la même récurrence que  $\binom{n}{p}$  avec les mêmes conditions initiales, donc  $t'(n, p) = \binom{n}{p}$  et par suite  $t(n, p) = \binom{n}{p} - 1$ . Ce programme n'est pas très efficace.

**Exercice 16 (Suites de Fibonacci)** 1. Un premier programme «naïf» :

```
fibol := proc(n :: integer)
  local u, v, bol, j:
  u:=1:
  v:=1:
  if n <= 1 then 1
  else
```

```

        for j from 2 to n do
            bol := u:
            u := u+v:
            v:= bol:
        od:
    fi :
    u;
end:

```

Un second, récursif :

```

fibo2 := proc(n :: integer):
    if n <= 1 then 1
    else fibo2(n-1) + fibo2(n-2);
    fi :
end:

```

2. Soit  $v$  une suite constante solution de

$$c(n) = c(n-1) + c(n-2) + 1 \quad (1)$$

C'est  $v = -1$ . L'équation caractéristique de l'équation homogène est  $r^2 - r - 1 = 0$  de racines  $\frac{1-\sqrt{5}}{2}$  et  $\frac{1+\sqrt{5}}{2}$  donc  $u_n$  est de la forme

$$u_n = \alpha \left( \frac{1-\sqrt{5}}{2} \right)^n + \beta \left( \frac{1+\sqrt{5}}{2} \right)^n$$

donc

$$c(n) = u_n - 1$$

avec

$$\begin{cases} \alpha + \beta - 1 = 0 \\ \alpha \frac{1-\sqrt{5}}{2} + \beta \frac{1+\sqrt{5}}{2} - 1 = 0 \end{cases}$$

soit

$$c(n) = \frac{1}{2} \left( 1 - \frac{\sqrt{5}}{5} \right) \left( \frac{1-\sqrt{5}}{2} \right)^n + \frac{1}{2} \left( 1 + \frac{\sqrt{5}}{5} \right) \left( \frac{1+\sqrt{5}}{2} \right)^n - 3$$

3. Premier programme : si  $n = 0$  ou  $n = 1$ , il n'y a aucune opération. Si  $n \geq 2$  il y a  $n - 1$  boucles, chaque boucle contenant une addition (et

trois affectations). Donc le coût est :  $c(0) = 0$  et  $c(n) = n - 1$  si  $n \geq 1$ .  
Donc le coût est «linéaire» :

$$c(n) = O(n)$$

Second programme : le coût  $c$  vérifie  $c(0) = 0$  et  $c(1) = 0$  car 1 est renvoyé sans calcul. Pour  $n \geq 2$ ,  $c(n) = c(n-1) + c(n-2) + 1$  car l'appel à `fib2(n)` provoque deux appels `fib2(n-1)`, `fib2(n-2)` et nécessite leur addition. Donc le coût est exponentiel :

$$c(n) = O\left(\left(\frac{1 + \sqrt{5}}{2}\right)^n\right)$$

4. Pour ce programme récursif, qui calcule aussi les nombres de Fibonacci, le coût est identique au coût du premier programme, il est linéaire.

**Exercice 17** Voici une première version

```

puiss := proc(x, n :: integer)
local p, k:
p := 1:
if n <= 0 then 1 else
  for k to n do
    p := p * x:
  od:
fi:
p;
end:

```

puis une version récursive :

```

puiss_rec := proc(x, n :: integer):
  if n <= 0 then 1 else x * puiss_rec(x, n-1): fi;
end:

```

Leurs complexités en temps sont identiques, de l'ordre de  $O(n)$ .

En revanche, le programme suivant, qui effectue la même tâche, a une complexité en temps bien meilleure :

```

puiss_rapid := proc(x, n :: integer):
  if n <= 0 then 1
    elif n mod 2 = 0 then

```

```

        puiss_rapid(x, n/2)**2
    else
        x * puiss_rapid(x, (n-1)/2)**2
    fi ;
end :

```

Complexité : nous avons deux cas

1.  $n$  est pair,  $n = 2m$  :  $c(n) = c(2m) = c(m) + 1$  ;
2.  $n$  est impair,  $n = 2m + 1$  :  $c(2m + 1) = c(m) + 2$ .

Dans tous les cas :  $c(n) \leq c(\lfloor \frac{n}{2} \rfloor) + 2$ . Soit  $k = \log_2(n)$ , nous avons

$$2^k \leq n < 2^{k+1}$$

donc, pour tout entier  $j$  :

$$\lfloor \frac{n}{2^j} \rfloor = 2^{k-j}$$

et

$$c\left(\lfloor \frac{n}{2^j} \rfloor\right) \leq c\left(\lfloor \frac{n}{2^{j+1}} \rfloor\right) + 2$$

ce qui implique

$$c(n) \leq 1 + \sum_{j=0}^{k-1} 2 = 2k + 1$$

Autrement dit :

$$c(n) = O(\ln(n))$$

**Exercice 18** Un programme de conversion en binaire : ce programme suivant donne la suite des poids  $b_m, \dots, b_0$  :

```

binaire := proc(n :: integer)
local L:
L := NULL:
if n<0 then error "nombre non positif"
elif n = 0 then L
else binaire(floor(n/2)),n mod 2,L
fi :
end :

```

Un autre qui renvoie une liste :

```

binaire := proc(n :: integer);
  if n<=0 then [] else
    [op(binaire(n/2-(n mod 2)/2)), n mod 2]
  fi;
end;

```

**Exercice 19** Un programme de conversion en binaire, par exemple :

```

binaire := proc(n :: integer);
  if n<=0 then [] else
    [op(binaire(n/2-(n mod 2)/2)), n mod 2]
  fi;
end;

```

Un programme récursif auxiliaire :

```

pr_aux := proc(x,y,b);
  if b=[] then y elif b[1]=0 then pr_aux(x,y^2,b[2..nops(b)])
  else pr_aux(x,x*y^2,b[2..nops(b)])
  fi;
end;

```

Le programme principal (récursif terminal) :

```

puiss_rapide := proc(x, n :: integer);
  pr_aux(x,1,binaire(n))
end;

```

**Autres versions** : Développement binaire d'un entier.

```

bin := proc(n, b);
  if n <= 0 then b
  else bin(iquo(n, 2), [irem(n, 2), op(b)])
  end if;
end proc;

```

bin(5, []);

bin(n, []);.

Programme auxiliaire

```

pr := proc(x, l, n);
  if l = [] then n
  elif l[1] = 0 then pr(x, subsop(1 = NULL, l), n^2)
  else pr(x, subsop(1 = NULL, l), n*x)

```

```

    end if;
end proc;

```

```

pr(2,[1,1,0],2);

```

Pour obtenir  $x$  puissance  $n$  :  $L:=\text{bin}(n,[])$ ; puis,  $\text{pr}(x,L,x)$ ; . Ce qui donne finalement le programme suivant :

```

puissance_rapide := proc(x, n)
    local L;
    L := bin(n, []);
    pr(x, L, x);
end proc;

```

**Exercice 20 (Complexité des suites de Fibonacci) Question 1** En séance d'informatique nous avons vu un programme récursif et un programme itératif, de même complexité, en  $O(n)$ .

**Question 2** Effectuons une démonstration par récurrence sur  $n = p + q$ .  $n$  est supérieur ou égal à 1.

Si  $n = 1$  :  $u_{p+q} = u_1 = u_1u_1 + u_0u_0 = 1$  (la condition initiale  $u_0 = 0$  est importante).

Supposons la relation valable pour tous  $(p, q)$  tels que  $p + q = n$  et  $q \geq 1$ .

Soient  $p$  et  $q$  tels que  $p + q = n + 1$  et  $q \geq 1$ . Par définition :

$$u_{p+q} = u_{p-1+q} + u_{p-2+q}$$

puisque  $q \geq 1$ , d'après l'hypothèse de récurrence

$$\begin{aligned} u_{p+q} &= (u_p u_q + u_{p-1} u_{q-1}) + (u_{p-1} u_q + u_{p-2} u_{q-1}) \\ &= (u_p + u_{p-1}) u_q + (u_{p-1} + u_{p-2}) u_{q-1} \\ &= u_{p+1} u_q + u_p u_{q-1} \end{aligned}$$

donc la relation est vraie pour tout couple d'entiers positifs avec  $q \geq 1$ .

**Question 3** Appliquons la relation à  $p = m$  et  $q = m$ , nous devons donc supposer que  $m \geq 1$  :

$$u_{2m} = u_{m+1} u_m + u_m u_{m-1}$$

or  $u_{m-1} = u_{m+1} - u_m$

$$\begin{aligned} u_{2m} &= u_{m+1}u_m + u_m(u_{m+1} - u_m) \\ &= 2u_mu_{m+1} - u_m^2 \end{aligned}$$

Relation suivante :

$$\begin{aligned} u_{2m+1} &= u_{m+(m+1)} \\ &= u_{m+1}u_{m+1} + u_mu_m \end{aligned}$$

d'où la seconde relation. Pour la troisième :

$$\begin{aligned} u_{2m+2} &= u_{2m+1} + u_{2m} \\ &= 2u_{m+1}u_m - u_m^2 + u_{m+1}^2 + u_m^2 \\ &= 2u_{m+1}u_m + u_{m+1}^2 \end{aligned}$$

**Question 4** Si  $n = \sum_{k=0}^{q-1} 2^k a_k$  est strictement positif, sa représentation binaire en liste  $[a_q, \dots, a_0]$  est donnée par le programme récursif suivant :

$$\text{binaire}(n) := (a := n \bmod 2) \text{ dans } [\text{binaire}((n - a)/2), a]$$

qui correspond à l'identification  $[a_q, \dots, a_0] = [[a_q, \dots, a_1], a_0]$ .

```

binaire := proc(n)
  if n <= 0 then [ ]
  else [op(binaire(n/2 - (n mod 2)/2), n mod 2)]
  end if
end proc;

```

**Question 5** Pour  $k = q$ , posons  $m_q = a_q$  et pour tout entier  $k$  tel que  $0 \leq k \leq q - 1$  :  $m_k = 2m_{k+1} + a_k$ . Alors :  $R(m_{k+1}, [a_k, \dots, a_0])$  appelle  $R(2m_{k+1} + a_k, [a_{k-1}, \dots, a_0]) = R(m_k, [a_{k-1}, \dots, a_0])$ . Donc pour  $k = 0$  :

$$R(m_1, [a_0]) = R(2m_1 + a_0, []) = R(m_0, []) = m_0$$

Or pour tout  $k$ ,  $m_k - 2m_{k+1} = a_k$  donc  $2^k m_k - 2^{k+1} m_{k+1} = 2^k a_k$ , nous en déduisons, avec la convention que  $m_{q+1} = 0$  :

$$\begin{aligned} m_0 &= \sum_{0 \leq k \leq q} 2^k m_k - 2^{k+1} m_{k+1} \\ &= \sum_{0 \leq k \leq n} 2^k a_k \\ &= n \end{aligned}$$

Donc l'appel  $R(0, L) = R(m_{q+1}, [a_q, \dots, a_0])$  renvoie  $n$ .  
 Par exemple  $R(0, [1, 0, 1])$  donne 5.

### Question 6

*#calcul des termes de la suite*

```
fib_log := proc(u, v, b);
  if b=[ ] then u, v elif b[1] = 0 then
    fib_log(2*u*v-u^2, u^2+v^2, b[2..nops(b)]);
  else
    fib_log(u^2+v^2, 2*u*v+v^2, b[2..nops(b)]);
  fi;
end;
```

*#Programme principal*

```
F:=proc(n :: integer);
  fib_log(0,1,binaire(n));
end;
```

*#Exemples*

```
seq (F(0,1,2*i), i=0..10);
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233,
377, 610, 987, 1597, 2584, 4181, 6765, 10946
```

$$P(u_m, u_{m+1}, [a_k, \dots]) \rightarrow P(u_{2m+a_k}, u_{2m+a_k+1}, [a_{k+1}, \dots])$$

Soit

$$\begin{cases} \text{si } a_k = 0 & \text{alors } P(u_{2m}, u_{2m+1}, [a_{k+1}, \dots]) \\ \text{sinon} & P(u_{2m+1}, u_{2m+2}, [a_{k+1}, \dots]) \end{cases}$$

Preuve de l'algorithme.

Soit  $n = \sum_{k=0}^{k=q} 2^k a_k$  le développement binaire d'un nombre entier  $n$  strictement positif. Nous avons  $a_q = 1$ , posons  $a_{q+1} = 0$  donc  $1 \leq k \leq q+1$ .

Effectuons une récurrence descendante : au rang  $k = q+1$  nous avons  $\text{fib\_log}(u_0, u_1, [a_q, \dots])$ .

Hypothèse de récurrence le  $q+1 - k^e$  appel est :

$$\text{fib\_log}(u_{\sum_{j=k}^{j=q+1} 2^{j-k} a_j}, u_{(\sum_{j=k}^{j=q+1} 2^{j-k} a_j)+1}, [a_{k-1}, \dots])$$

Alors le  $q + 1 - (k + 1)^e$  est (si  $k \geq 2$ ) :

$$\text{fibonacci\_log}(u_{2^{(\sum_{j=k}^{j=q+1} 2^{j-k} a_j) + a_{k-1}}}, u_{2^{(\sum_{j=k}^{j=q+1} 2^{j-k} a_j) + a_{k-1} + 1}}, [a_{k-2}, \dots])$$

$$\text{fibonacci\_log}(u_{\sum_{j=k-1}^{j=q+1} 2^{j-(k-1)} a_j}, u_{\sum_{j=k-1}^{j=q+1} 2^{j-(k-1)} a_j + 1}, [a_{k-2}, \dots])$$

Par suite, le  $q = q + 1 - 1^e$  donne

$$\text{fibonacci\_log}(u_{\sum_{j=1}^{j=q+1} 2^{j-1} a_j}, u_{(\sum_{j=1}^{j=q+1} 2^{j-1} a_j) + 1}, [a_0])$$

et le dernier :

$$\text{fibonacci\_log}(u_{\sum_{j=0}^{j=q+1} 2^j a_j}, u_{(\sum_{j=0}^{j=q+1} 2^j a_j) + 1}, []) = \text{fibonacci\_log}(u_n, u_{n+1}, [])$$

qui renvoie  $u_n, u_{n+1}$ .

**Question 7** Le nombre d'appels récursifs lorsque  $n = 2^m$  est  $m$ , soit  $\ln_2 n$ , la complexité est en  $O(\ln n)$  puisque le calcul de l'écriture binaire de  $n$  est aussi du même ordre.

## 9.4 Tris

**Exercice 21** Rappelons les commandes qui donnent des listes d'entiers à trier

```
d:=rand(100); L:=[seq(d(), j=1..10)];
```

**Question 1 (Tri sélection)** pour  $i$  de 1 à  $N - 1$  faire

$m \leftarrow i$  «  $m$  est le rang du plus petit » «  $a_1, \dots, a_{i-1}$  sont triés »

pour  $j$  de  $i + 1$  à  $N$  faire

si  $a_j < a_i$  alors

$m \leftarrow j$  « recherche du plus petit »

fin si

fin pour

$t \leftarrow a_m$ ;  $a_m \leftarrow a_i$ ;  $a_i \leftarrow t$

fin pour

```
selection := proc(L)
```

```
local a, i, j, m, t;
```

```
a := L;
```

```
n := nops(a);
```

```
for i from 1 to n-1 do
```

```
  m := i;
```

```
  for j from i+1 to n do
```

```
    if a[j] < a[m] then m := j fi;
```

```
  od;
```

```
  t := a[m];
```

```
  a[m] := a[i];
```

```
  a[i] := t;
```

```
od;
```

```
a;
```

```
end;
```

**Question 2 (Tri insertion)** pour  $i$  de 2 à  $n$  faire

«  $n$  est le nombre d'éléments à trier »

$m \leftarrow a_i$  «  $m$  est le terme comparé aux précédents »  $j \leftarrow i - 1$  «  $a_1, \dots, a_{i-1}$  sont triés »

tant que  $j \geq 1$  et  $a_j > m$  faire

$a_{j+1} \leftarrow a_j$  « comparaison, décalage »

$j \leftarrow j - 1$  « terme suivant »

fin tant que

```

    si  $j \geq 0$  alors
         $a_{j+1} \leftarrow v$  «  $a_i$  est placé »
    fin si
fin pour

insere := proc(M)
    local i, j, v, n, L;
    L := M;
    n := nops(L);
    for i from 2 to n do
        v := L[i];
        j := i - 1;
        while 1 <= j and v < L[j] do
            L[j + 1] := L[j]; j := j - 1
        end do;
        L[j+1] := v;
    print(L);
    end do;
end proc;

```

**Question 3 (Tri bulle)** « parcours »

```

pour j de 1 à  $n - 1$  faire
    « comparaisons »
    pour k de 1 à  $n - j$  faire
        si  $a_{k+1} < a_k$  alors
             $b \leftarrow a_k$  « échange des éléments »
             $a_k \leftarrow a_{k+1}$ 
             $a_{k+1} \leftarrow b$ 
        fin si
    fin pour
fin pour

bulle := proc(L)
local j, k, n, b, M;
M := L;
n := nops(M);
for j to  $n - 1$  do
    for k to  $n - j$  do
        if  $M[k + 1] < M[k]$  then
            b := M[k];
            M[k] := M[k + 1];

```

```

        M[k + 1] := b
    end if
end do
end do
M;
end proc

```

Exemples

```
bulle ([5, 6, 1, 2, 8, 4, 3]);
```

```
    [1, 2, 3, 4, 5, 6, 8]
```

```
bulle ("s", "o", "p", "h", "i", "a", "a", "n", "t", "i", "p", "o", "l", "i", "s");
    ["a", "a", "h", "i", "i", "i", "l", "n", "o", "o", "p", "p", "s", "s", "t"]
```

**Question 4 (Tri fusion)**  $\ell, m$  et  $n$  sont les nombres d'éléments de  $L, M$  et  $N$  respectivement.

« si l'une des listes  $M$  ou  $N$  est vide, on renvoie » « la liste  $[op(L), op(M), op(N)]$ , c'est une liste triée »

**si**  $m = 0$  ou  $n = 0$  **alors**

alors juxtaposer  $L, M$  et  $N$ , dans cet ordre

**ou si**  $M[1] \leq N[1]$  **alors**

$L \leftarrow [L[1], \dots, L[\ell], M[1]]$

$M \leftarrow [M[2], \dots, M[m]]$

appeler la procédure avec les nouvelles valeurs  $L, M, N$

**sinon**

$L \leftarrow [L[1], \dots, L[\ell], N[1]]$

$N \leftarrow [N[2], \dots, N[n]]$

appeler la procédure avec les nouvelles valeurs  $L, M, N$

**fin si**

```
fusion := proc (L, M, N)
```

```
    local m, n:
```

```
    m := nops(M):
```

```
    n := nops(N):
```

```
    if m=0 or n=0 then [op(L), op(M), op(N)]
```

```
        elif M[1] <= N[1] then fusion ([op(L), M[1]], M[2..m], N)
```

```
        else fusion ([op(L), N[1]], M, N[2..n])
```

```
    fi;
```

```
end;
```

$M$  et  $N$  étant triées, l'appel `fusion ([], M, N)` fusionne les deux listes en une liste triée.

**Exemple 9.1** `fusion([], [1, 3, 5, 7, 9], [2, 4, 6, 8]);`  
`[1, 2, 3, 4, 5, 6, 7, 8]`

Comment partitionner :

```
partition := proc(L)
  local n, M, N;
  n := nops(L);
  if n <= 1 then L else
    M, N := [op(L[1.. floor(n/2)])], [op(L[ floor(n/2)+1..n])];
  fi;
end;
```

Version récursive du tri fusion :

```
trifusion := proc(L) local n, L1, L2;
  n := nops(L);
  if n <= 1 then L else
    L1, L2 := partition(L);
    fusion([], trifusion(L1), trifusion(L2));
  fi;
end;
```

**Question 5 (Tris rapide)** Le tri a un intérêt seulement s'il y a plus de 3 éléments, ce que nous supposons.

La description du tri rapide est faite dans le cours.

Procédure de partition :

```
partition := proc(liste, k, m)
  local L, i, j, pivot, bol;
  L := liste;
  pivot := L[m]; i := k; j := m-1;
  if i <> j then
    while i < j do
      while L[i] <= pivot and i < m do i := i+1; od;
      while L[j] >= pivot and j > k do j := j-1; od;
      if i < j then bol := L[j]; L[j] := L[i]; L[i] := bol; fi;
    od;
    L[m] := L[i]; L[i] := pivot;
    elif i = j and L[k] > L[m] then L[m] := L[k]; L[k] := pivot;
  fi;
  L, i;
```

```
end:
    Tri rapide :
trirap := proc (liste , i , j)
    local k,L:
    L:= liste :
    if i<j then partition(L,i , j):
        L:= %[1];
        k:= %%[2];
        L:= trirap (L,i ,k-1);
        L:= trirap (L,k+1,j);
    fi;
    L;
end:
```

**Exercice 22** Tri sélection :

Complexité au pire : le programme fait appel à deux fonctions, `plus_petit` et `retire` dont les coûts sont proportionnels à la longueur de la liste. Comme `retire` comporte un élément de moins, on a, si  $n$  est la longueur de la liste :  $c(n) = an + c(n - 1)$ . D'où :

$$\Delta c(n) = c(n) - c(n - 1) = an$$

Avec  $c(0) = 0$  on déduit :

$$c(n) = \sum_{1 \leq k \leq n} ak = a \frac{n(n+1)}{2}$$

soit une complexité en  $O(n^2)$ .

Tri insertion :

Complexité au pire : on utilise deux fonctions `insere` et `tri_inser`. La fonction `insere` a un coût proportionnel à la longueur  $n$  de la liste. `tri_inser` sur  $l$  appelle `tri_inser` sur une liste de longueur  $n - 1$  puis `insere` sur cette liste triée de longueur  $n - 1$  d'où :

$$c(n) = a(n - 1) + c(n - 1)$$

D'où une complexité en  $O(n^2)$ . On prend habituellement  $a = 1$  pour normaliser, ce qui ne change pas l'ordre de grandeur de la complexité.

Tri rapide :

Au pire la complexité de partition est  $O(n)$  et au mieux le pivot est toujours au milieu, ce qui donne pour  $n = 2^m$  (1 est le coût du premier appel) :

$$c(2^m) = 1 + c(2^{m-1})$$

d'où  $c(2^m) = m$  et une complexité en  $\log_2 n$ .

Soit  $C(n)$  le coût de `tri_rapide` :

$$C(n) = c(n) + 2C(n/2)$$

Posons  $u_m = 2^{-m}C(2^m)$ , il vient :

$$u_m = 2^{-m}c(2^m) + u_{m-1}$$

soit

$$u_m - u_{m-1} = 2^{-m}m$$

d'où

$$u_m = \sum_{0 \leq k \leq m} 2^{-k} k$$

La somme est calculée à partir de la dérivée de  $\sum_{0 \leq k \leq m} x^{k+1} = x \frac{1-x^{m+1}}{1-x}$  qui vaut  $\sum_{0 \leq k \leq m} kx^k = \frac{1-x^{m+2}}{(1-x)^2} - \frac{(m+2)x^{m+1}}{1-x}$ . Donc  $u_m$  est un  $O(m)$  et  $C(2^m)$  un  $O(m2^m)$ , finalement :  $C(n) = O(n \log_2 n)$ . Dans le pire des cas, le pivot est toujours mal placé (début ou fin de liste) et la complexité est un  $O(n^2)$ .

Tri fusion :

Le calcul de la complexité du tri fusion est semblable au calcul de la complexité du tri rapide dans le meilleur des cas. La complexité est donc un  $O(n^2)$ .