INFORMATIQUE 1ère année

"*Enfin, cher lecteur, [...] j'espère que la seule pensée à trouver une troisième méthode pour faire toutes les opérations arithmétiques, totalement nouvelle et qui n'a rien de commun avec les deux méthodes vulgaires de la plume et du jeton, recevra de toi quelque estime et qu'en approuvant le dessein que j'ai eu de te plaire en te soulageant, tu me sauras gré du soin que j'ai pris pour faire que toutes les opérations, qui par les précédentes méthodes sont pénibles, composées, longues et peu certaines, deviennent faciles, simples, promptes et assurées.*"
Blaise Pascal - La Machine Arithmétique.

PLAN

I : Algorithmes

1- Actions élémentaires

Le mot algorithme provient du nom du mathématicien arabe Al Kharezmi, inventeur de l'algèbre, né durant le IXème siècle en Perse. Un algorithme est une suite finie d'instructions à appliquer dans un

ordre déterminé dans le but de résoudre un problème donné. Chacun de nous applique les algorithmes appris dans l'enfance lorsqu'il calcule la somme de deux nombres, leur produit ou leur quotient.

Les algorithmes, aussi complexes soient-ils, sont construits à partir d'actions élémentaires, essentiellement au nombre de trois :

- les affectations de variables
- les instructions conditionnelles
- les instructions itératives

A cela, il faut ajouter les instructions de lecture des données et de sortie des résultats. Nous utiliserons une notation symbolique, adaptable à n'importe quel langage de programmation. Nous donnerons également des exemples de traduction syntaxique d'un algorithme en un programme utilisable sous **Python**, langage de programmation, **Scilab**, logiciel dédié au calcul numérique de données matricielles, **Maple**, logiciel de calcul formel, et **Java**, langage de programmation. Cependant, ceci n'est pas un cours d'apprentissage d'un de ces langages ou logiciels, mais un cours généraliste sur les notions universelles rencontrées en informatique. Le lecteur peut également transcrire les algorithmes utilisés les plus simples sur sa calculatrice programmable ou en n'importe quel autre langage de programmation.

2- Affectations de variables

L'affectation de variable permet d'attribuer des valeurs à des variables ou de changer ces valeurs. Les variables sont représentées par un nom, qui peut comporter plusieurs lettres. La plupart des langages de programmation modernes font la distinction entre majuscules et minuscules. Nous symboliserons l'affectation de variable de la façon suivante :

Y ← 2	Y prend la valeur 2
X ← 2*Y+1	puis X la valeur 5 (* désigne le produit)
X ← 3*X +2	puis X prend la valeur 17

Le membre de droite est une expression calculée par la machine, puis, une fois ce calcul effectué, le résultat est stocké dans la variable figurant dans le membre de gauche. Si le même nom figure dans les deux membres, cela signifie que la variable change de valeur au cours du calcul. Dans la dernière instruction ci-dessus, l'ancienne valeur 5 de X est définitivement perdue au cours de l'exécution du calcul et remplacée par la valeur 17.

Affectation de variables	
En Python	En Scilab
Y = 2 X = 2*Y + 1 X = 3*X + 2	Y = 2 X = 2*Y + 1 X = 3*X + 2
En Maple	En Java
Y := 2; X := 2*Y + 1; X := 3*X + 2;	Y = 2; X = 2*Y + 1; X = 3*X + 2;

Le choix du symbole "=" en Python, Scilab et Java n'est pas très heureux, car l'instruction "←" ne désigne pas une égalité mathématique, mais une action visant à changer la valeur d'une variable. Le choix de ":=" dans Maple est de ce point de vue plus clair.

Le changement simultané de deux variables demande une certaine attention. Supposons qu'on dispose d'un couple (a, b) dont la valeur a été préalablement assignée et qu'on veuille changer la valeur de ce couple en (b, $a + b$). La commande suivante est incorrecte :

a ← b
b ← a + b

car dans la deuxième instruction, la valeur de a figurant dans le membre de droite a été modifiée en celle de b dans la première instruction, ce qui a contribué à effacer la précédente valeur de a. A la fin du calcul, on a en fait remplacé le couple (a, b) par le couple (b, $2b$). De même :

b ← a + b
a ← b

donne la valeur correcte de b, mais recopie ensuite cette valeur dans a, de sorte que le couple (a, b) a été remplacé par le couple ($a + b$, $a + b$). Il convient d'utiliser une variable temporaire. Notons (a_0, b_0) la valeur initiale du couple (a, b). On indique en commentaire les valeurs de chaque variable au cours du calcul. Il est utile d'ajouter ce genre de commentaire dans un programme afin d'en prouver la validité. Les commentaires sont précédés d'un # en Python et Maple, d'un // en Scilab ou Java.

tmp ← b # après cette instruction, $tmp = b_0$, $a = a_0$, $b = b_0$
b ← a + b # après cette instruction, $tmp = b_0$, $a = a_0$, $b = a_0 + b_0$
a ← tmp # après cette instruction, $tmp = b_0$, $a = b_0$, $b = a_0 + b_0$

On a bien le résultat attendu. On aurait pu faire :

tmp ← a # après cette instruction, $tmp = a_0$, $a = a_0$, $b = b_0$
a ← b # après cette instruction, $tmp = a_0$, $a = b_0$, $b = b_0$
b ← tmp + b # après cette instruction, $tmp = a_0$, $a = b_0$, $b = a_0 + b_0$

Dans les deux cas, c'est la variable dont on a stocké la valeur dans tmp qu'on modifie en premier.

De même, si on veut permuter les valeurs de deux variables, on procèdera comme suit :

tmp ← a # après cette instruction, $tmp = a_0$, $a = a_0$, $b = b_0$
a ← b # après cette instruction, $tmp = a_0$, $a = b_0$, $b = b_0$
b ← tmp # après cette instruction, $tmp = a_0$, $a = b_0$, $b = a_0$

Signalons que Python dispose d'une option d'affectation simultanée des variables évitant l'utilisation de la variable *tmp* :

```
a,b = b,a + b
a,b = b,a
```

3- Instruction conditionnelle

Nous écrirons cette instruction :

```
si <Condition>
        alors <Bloc d'Instructions 1>
        sinon <Bloc d'Instructions 2>
finsi
```

<Bloc d'Instructions 1> désigne une ou plusieurs instructions à exécuter si <Condition> est vraie. <Bloc d'Instructions 2> désigne une ou plusieurs instructions à exécuter si <Condition> est fausse. Par exemple :

```
si X >0
        alors X ← X -1
        sinon X ← X + 1
finsi
```

Cette instructions retranche 1 à une variable X positive et ajoute 1 à une variable négative ou nulle.

Dans le cas où il n'y a pas de <Bloc d'Instructions 2> à exécuter, on mettra l'instruction conditionnelle sous la forme :

```
si <Condition>
        alors <Blocs d'Instructions 1>
finsi
```

<Condition> est une expression booléenne pouvant prendre la valeur vraie (True) ou fausse (False), telle que X = 0, X > 0, X ≥ 0, X ≠ Y, X est un entier pair, etc... La façon de transcrire les expressions booléennes est propre à chaque langage. Par exemple, Les quatre conditions ci-dessus se traduisent par :

Expressions booléennes	
En Python	En Scilab
X == 0 X > 0 X >=0 X != Y X%2 == 0	X == 0 X > 0 X >=0 X <> Y modulo(X,2) == 0
En Maple	En Java
X = 0 X > 0 X >=0 X <> Y X mod 2 == 0	X == 0 X > 0 X >=0 X != Y X%2 == 0

On notera l'utilisation d'un double symbole = = pour tester l'égalité de deux variables dans les langages Python, Scilab et Java puisque ces langages utilisent déjà le simple = pour l'affectation de variable.

La façon de coder l'instruction conditionnelle et en particulier de signaler la fin de chaque bloc d'instructions, est également propre à chaque langage. En Python, la limite des blocs d'instructions est donnée par simple indentation, c'est-à-dire par un retrait plus ou moins prononcé à partir de la marge de gauche. Les autres langages ont des marqueurs typographiques de paragraphes plus visibles. Néanmoins, même dans ces derniers, l'indentation est vivement recommandée pour des raisons de lisibilité.

<table>
<tr><th colspan="2">Instruction conditionnelle</th></tr>
<tr><th>En Python</th><th>En Scilab</th></tr>
<tr><td>if condition:
  Bloc d'Instructions 1
else:
  Bloc d'Instructions 2</td><td>if condition then
  Bloc d'Instructions 1
else
  Bloc d'Instructions 2
end</td></tr>
<tr><th>En Maple</th><th>En Java</th></tr>
<tr><td>if condition then
  Bloc d'Instructions 1
else
  Bloc d'Instructions 2
fi;</td><td>if (Condition)
  {Bloc d'Instructions 1}
else {Bloc d'Instructions 2}</td></tr>
</table>

En Python, noter les : en bout de lignes contenant le if et le else.

4- Expressions booléennes

Les expressions booléennes intervenant dans la condition d'une instruction conditionnelle peuvent être combinées entre elles, comme en mathématiques, au moyen des opérateurs de conjonction (et), de disjonction (ou) et de négation (non). La disjonction est prise au sens large, c'est-à-dire que "A ou B" est vraie à partir du moment où une seule des deux propositions est vraie. Il existe également deux expressions atomiques True et False, l'une ayant la valeur "Vrai", l'autre ayant la valeur "Faux". La traduction de ces opérateurs diffèrent d'un langage à l'autre. On peut affecter une expression booléenne à une variable (B dans les exemples ci-dessous).

Opérateurs booléens	
En Python	En Scilab
True False (X>0) or (Y==0) B = (X<=0) and (Y!=0) not B	%T %F (X>0) | (Y==0) B = (X<=0) & (Y<>0) ~B
En Maple	En Java
true false (X>0) or (Y=0) B := (X<=0) and (Y<>0) not(B)	True False (X>0) | (Y==0) B = (X<=0) & (Y!=0) !B

Si les connecteurs logiques "et", "ou" et "non" ont le même sens dans tous les langages, leur évaluation effective peut différer. Considérons deux variables numériques X et Z. On initialise X à 1, mais Z reste non initialisée. Considérons maintenant l'expression booléenne A suivante :

A ← (X > 0) ou (Z = 1)

Quelle valeur faut-il lui attribuer ? Les logiciels diffèrent sur ce point. Scilab remarquera que Z n'a pas été initialisée, donc qu'il est impossible de définir la valeur de vérité de A. Le logiciel s'arrête alors à cette instruction en indiquant une erreur due au défaut d'initialisation de Z. Python, pour des raisons de rapidité, évaluera d'abord l'expression X > 0, notera qu'elle est vraie puis en déduira que A est vraie quelle que soit la valeur de vérité de l'expression Z = 1 et donc ne cherchera pas à évaluer cette dernière. La valeur de vérité True sera attribuée à A et l'exécution du programme se poursuivra. Cependant, si X avait été initialisée à – 1, il aurait évalué la valeur de vérité de Z = 1 et une erreur d'exécution se serait alors produite.

Pour des raisons de sûreté, il est prudent que le programmeur ne prévoit une évaluation booléenne que s'il est certain que chaque expression booléenne possède effectivement au moment de son exécution une valeur bien définie, faute de quoi un programme qui semble fonctionner dans certains cas pourrait soudainement cesser de fonctionner dans d'autres. Par exemple, rien ne dit qu'un logiciel particulier ne va pas commencer à évaluer les expressions booléennes de la droite vers la gauche plutôt que de la gauche vers la droite.

5- Instruction itérative

Cette instruction consiste à répéter un certain nombre de fois les mêmes instructions. C'est ce qui fait la puissance d'un programme informatique. On distingue deux types d'itérations :

❑ Si le nombre *n* d'itérations est connu à l'avance, on écrira :

```
pour i ← 1 à n faire
        <Bloc d'Instructions>
finfaire
```

i est une variable appelée compteur de boucles. Dans l'exemple précédent, <Bloc dInstructions> est exécuté *n* fois, *n* étant une variable préalablement définie. Par exemple, la boucle suivante calcule la somme des *n* premiers carrés d'entiers :

```
S ← 0
pour i ← 1 à n
        S ← S + i*i
finfaire
```

Plus généralement, on peut faire varier *i* entre deux valeurs données :

```
pour i ← deb à fin faire
        <Bloc d'Instructions>
finfaire
```

Instruction itérative	
En Python	En Scilab

for i in range(n):
Bloc d'Instructions
 	
for i in range(deb, fin+1):
Bloc d'Instructions
	for i in deb:fin
	Bloc d'Instructions
	end

En Maple	En Java
for i from deb to fin do *Bloc d'Instructions* od;	for (i=deb ; i <= fin ; i++) {*Bloc d'Instructions*}

On prendra garde qu'en Python, dans le premier exemple, *i* varie en fait de 0 à *n* – 1. range(n) désigne en effet une structure permettant de parcourt la suite des entiers naturels strictement inférieur à *n*. De même, pour parcourir les entiers entre deb et fin inclus, on utilisera range(deb, fin+1), la dernière valeur de range étant exclue. Comme pour l'instruction conditionnelle, c'est l'indentation qui marque la limite du bloc.

❑ Si le nombre d'itérations est inconnu et dépend de la réalisation d'une condition, on écrira :

```
tant que <Condition> faire
        <Bloc d'instructions>
finfaire
```

Par exemple, on calcule la plus petite puissance de 2 supérieure ou égale à la variable *n* comme suit, *n* étant un entier supérieur ou égal à 2 préalablement défini :

```
P ← 1
tant que P < n faire          # au début de la boucle, P est une puissance de 2 et P < n
        P ← P*2               # après cette instruction, P est une puissance de 2 et P/2 < n
finfaire                      # à la fin de la boucle, P est une puissance de 2 et P/2 < n ≤ P
```

Les commentaires apportent la preuve de la validité de l'algorithme. Le raisonnement est analogue à un raisonnement par récurrence. Initialement, P = 1 qui est bien une puissance de 2. Puis, si P est une puissance de 2 et voit sa valeur doubler, il reste une puissance de 2. Par ailleurs, si au début d'une boucle, on a P < *n* et si P voit sa valeur doubler, alors le prédicat vérifié après ce changement de valeur est désormais P/2 < *n* puisque la valeur actuelle P/2 représente alors l'ancienne valeur de

P. On sort de la boucle quand le prédicat P < *n* devient faux, i.e. quand P ≥ *n*. On a donc alors P/2 < *n* ≤ P ce qui prouve bien que P est la plus petite puissance de 2 supérieure ou égale à *n*. Ces explications, peut-être trop longues pour un cas simple, donnent néanmoins une première initiation à la technique de preuve d'un algorithme.

<table>
<tr><th colspan="2">Instruction itérative</th></tr>
<tr><th>En Python</th><th>En Scilab</th></tr>
<tr><td>while condition:
  Bloc d'instructions</td><td>while condition
  Bloc d'instructions
end</td></tr>
<tr><th>En Maple</th><th>En Java</th></tr>
<tr><td>while condition do
  Bloc d'instructions
od;</td><td>while (condition)
  {Bloc d'instructions}</td></tr>
</table>

6- Exemples

❑ <u>La suite de Collatz</u>

Collatz a étudié la suite définie par a_0 entier strictement positif et $a_{n+1} = \begin{cases} \frac{a_n}{2} \text{ si } a_n \text{ est pair} \\ 3a_n + 1 \text{ si } a_n \text{ est impair} \end{cases}$.

Si on part de 27, on obtient :

27, 82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161, 484, 242, 121, 364, 182, 91, 274, 137, 412, 206, 103, 310, 155, 466, 233, 700, 350, 175, 526, 263, 790, 395, 1186, 593, 1780, 890, 445, 1336, 668, 334, 167, 502, 251, 754, 377, 1132, 566, 283, 850, 425, 1276, 638, 319, 958, 479, 1438, 719, 2158, 1079, 3238, 1619, 4858, 2429, 7288, 3644, 1822, 911, 2734, 1367, 4102, 2051, 6154, 3077, 9232, 4616, 2308, 1154, 577, 1732, 866, 433, 1300, 650, 325, 976, 488, 244, 122, 61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, 4, 2, 1, etc...

On ignore aujourd'hui si, pour tout a_0, la suite finit par boucler par le cycle 1, 4, 2, 1 ou non. En 2020, cette propriété a été vérifiée pour tout $a_0 \leq 10^{20}$. On sait seulement que, s'il existe un cycle non trivial, il possède une période de longueur au moins égale à 17 milliards. Voici ci-dessous un algorithme affichant tous les termes de la suite jusqu'à ce qu'on arrive à 1, et calculant le nombre de termes affichés. Cette dernière valeur est stockée dans une variable nommée nbrIter. Le paramètre *a* est la valeur initiale de la suite, supposée connue au moment où l'on effectue ces instructions :

```
nbrIter ← 0
tant que a ≠ 1 faire
        si a pair alors a ← a/2
                sinon a ← 3*a + 1
        finsi
        Afficher a
        nbrIter ← nbrIter + 1
finfaire
```

Dans la pratique, il est utile de regrouper ces instructions en une fonction de la valeur initiale *a*, fonction à laquelle on peut donner un nom de son choix, Collatz par exemple, et dont le résultat est la valeur finale de nbrIter. On peut ensuite utiliser cette fonction comme une nouvelle commande en demandant Collatz(27) par exemple.

La suite de Collatz

En Python

```
def Collatz(a):
    nbrIter=0
    while a!=1:
        if a%2==0:
            a = a//2
        else:
            a = 3*a+1
        print(a)
        nbrIter+=1
    return(nbrIter)
```

En Scilab

```
function y=Collatz(a)
    nbrIter=0
    while a<>1
        if modulo(a,2)==0 then
                a = a/2
        else a = 3*a+1
        end
        disp(a)
        nbrIter = nbrIter+1
    end
    y = nbrIter
endfunction
```

En Maple

```
Collatz:=proc(a0)
  local a, nbrIter:
  a:=a0;
  nbrIter := 0;
  while a <> 1 do
        if a mod 2 = 0 then a := a/2
                else a := 3*a+1
        fi;
        print(a);
        nbrInter := nbrIter+1;
  od;
  nbrIter;
end:
```

En Java

```
int Collatz(int a0)
  {int a = a0;
   int nbrIter = 0;
   while (a != 1)
        {if (a % 2 == 0) {a = a/2;}
            else {a = 3*a+1;}
         System.out.println(a);
          nbrIter+=1;
        }
   return(nbrIter)
   }
```

On remarquera qu'en Python, le quotient de la division euclidienne dans les entiers se note //. Par ailleurs, on peut également noter la syntaxe abrégée pour ajouter un nombre à la variable nbrIter. Ceci permet d'accéléler l'exécution du programme en ne cherchant qu'une fois l'adresse en mémoire où se trouve la variable nbrIter.

En Maple, le paramètre a_0 ne peut voir sa valeur modifiée dans la fonction. C'est pourquoi cette valeur est copiée dans une variable *a* locale à la procédure. Il en est de même en Java. Par ailleurs, dans ce dernier langage, le type de chaque variable utilisée doit être clairement défini par l'utilisateur (type entier int dans le cas présent pour toutes les variables utilisées). En Python ou Maple, le type de la variable est implicitement définie par sa première affectation. En Scilab, le type par défaut est le type float des nombres réels. Les calculs sont a priori approchés, et dans certains cas, la parfaite exactitude du résultat peut ne pas être garantie.

❑ La multiplication égyptienne

Afin de multiplier deux nombres entre eux, par exemple 43 et 34, les égyptiens faisaient comme suit. Ils divisent l'un des nombres par 2, jusqu'à obtenir 1, l'autre étant multiplié par 2. On ajoute les multiples du deuxième nombre correspondant à des quotients impairs du premier. Cet algorithme n'est autre que l'algorithme usuel de multiplication, mais lorsque les nombres sont écrits en binaire :

```
43      34
21      68
10     136
 5     272
 2     544
 1    1088
 ————————
34 + 68 + 272 + 1088 = 1462 = 43 × 34
```

L'algorithme est le suivant. Nous avons besoin de trois variables, A, B et S. A et B prendront les valeurs successives calculées dans chaque colonne, S sera la somme partielle des valeurs B lorsque A sera impair. Nous commentons les instructions en notant entre parenthèses les relations vérifiées par les variables A, B et S au fur et à mesure du calcul. Pour cela, a_k, b_k et s_k désignent les valeurs de A, B et S après k boucles. Nous montrons alors par récurrence que AB + S est constante, égale à ab, produit des deux valeurs initiales. Cette relation constitue alors ce qu'on appelle un **invariant de boucle**. // désigne le quotient entier de deux entiers, à adapter à la syntaxe propre au langage utilisé.

```
A ← a                                  # initialement A = a
B ← b                                  # et B = b
S ← 0                                  # S = 0 donc ab = S + AB
tant que A ≠ 0 faire                   # Invariant de boucle : ab = s_k + a_k b_k
    si A impair alors S ← S+B finsi    # s_{k+1} = s_k  si a_k est pair
                                       # et s_{k+1} = s_k + b_k si a_k est impair
    A ← A // 2                         # a_{k+1} = a_k / 2 si a_k est pair
                                       # et a_{k+1} = (a_k – 1) / 2 si a_k est impair
    B ← B * 2                          # b_{k+1} = 2b_k  et dans tous les cas,
                                       # on a : s_{k+1} + a_{k+1}b_{k+1} = s_k + a_k b_k
    finfaire                           # On a donc bien s_{k+1} + a_{k+1}b_{k+1} = ab
                                       # et donc à nouveau ab = S + AB
```

On termine la boucle quand A = 0 ; le résultat final ab se trouve donc dans S. On aurait pu également apporter la preuve de la validité de l'algorithme en indiquant simplement après chaque instruction les relations entre A, B et S, ce qui est plus concis, mais peut-être plus difficile à comprendre. Par exemple, à la cinquième ligne ci-dessous, si A est impair, on augmente S de B, donc, si on avait la relation ab = S + AB avant cette instruction, on a nécessairement, après avoir augmenté S de B, ab = S – B + AB.

```
A ← a                                  # A = a
B ← b                                  # B = b
S ← 0                                  # S = 0 donc ab = S + AB
tant que A ≠ 0 faire                   # Invariant de boucle : ab = S + AB
```

```
si A impair alors S ← S+B finsi      # si A est pair,  ab = S + AB
                                     # si A est impair, ab = S – B + AB = S + (A – 1)B
A ← A // 2                           # Dans tous les cas, ab = S + 2AB
B ← B * 2                            # Dans tous les cas, ab = S + AB
finfaire                             # On sort de la boucle quand A = 0 donc ab = S
```

La multiplication égyptienne

En Python

```
def Egypt(a,b):
    A = a
    B = b
    S = 0
    while A != 0:
        if A % 2 == 1:
            S = S+B
        A = A//2
        B = B*2
    return(S)
```

En Scilab

```
function y=Egypt(a,b)
A = a
B = b;
S = 0;
while A <> 0
        if modulo(A,2) == 1 then
                S = S+B
        end
        A = floor(A/2)
        B = B*2
end
y = S
endfunction
```

En Maple

```
Egypt:=proc(a,b)
local A,B,S:
A := a;
B := b;
S := 0;
while A <> 0 do
    if A mod 2 = 1 then S:=S+B fi;
    A := iquo(A,2);
    B := B*2;
od;
S;
end:
```

En Java

```
int Egypt(int a, int b){
    int A = a, B = b, S = 0;
    while (A != 0) {
        if (A % 2 == 1) {S = S+B;}
        A = A/2;
        B = B*2;
        }
    return(S);
}
```

Si on remplace la somme par le produit et 0 par 1, alors le résultat de la procédure donne la valeur de $P = b^a$, l'invariant de boucle étant alors $b^a = P*B^A$.

```
A ← a                                # A = a
B ← b                                # B = b
P ← 1                                # P = 1 donc b^a = P*B^A
tant que A ≠ 0 faire                 # Invariant de boucle : b^a = P*B^A
    si A impair alors P ← P*B finsi  # si A est pair,  b^a = P*B^A
                                     # si A est impair, b^a = P/B*B^A = P*B^(A–1)
    A ← A // 2                       # Dans tous les cas, b^a = P*B^2A
    B ← B * B                        # Dans tous les cas, b^a = P*B^A
    finfaire                         # On sort de la boucle quand A = 0 donc b^a = P
```

Cela permet de calculer la puissance b^a en O(ln(a)) opérations au lieu de O(a). Cet algorithme est connu sous le nom d'exponentiation rapide.
Si a, b et n sont des entiers strictement positifs, avec $2 \leq b < n$, et si on remplace dans l'algorithme les produits P∗B et B∗B par P∗B mod n et B∗B mod n, on obtient un moyen efficace de calculer b^a mod n, y compris pour des valeurs de a très grandes, les valeurs de P et B restant limitées par n. A l'inverse, le calcul direct de b^a avant de prendre le reste modulo n peut conduire à des nombres extrêmement grands (en supposant encore que le langage de programmation choisi puisse manipuler de grands entiers) risquant d'encombrer la mémoire de la machine. Certains langages de programmation possèdent des syntaxes particulières permettant de calculer b^a mod n en évitant ce passage par de très grands nombres.

II : Types de données

1- Le stockage de l'information

La plus petite information utilisable dans un ordinateur est le chiffre binaire (*binary digit* ou *bit*) 0 ou 1. Ces chiffres binaires sont regroupés par 8 pour donner un octet. Il existe donc $2^8 = 256$ octets, depuis 0000 0000 jusqu'à 1111 1111. La moitié d'un octet est représenté par 4 chiffres binaires, donnant $2^4 = 16$ combinaisons possibles, depuis 0000 jusqu'à 1111. Ces combinaisons sont représentées par les symboles 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F appelés chiffres hexadécimaux. Un octet peut donc être également représenté par deux chiffres hexadécimaux. Par exemple :

0101 1100	ou	5C
0101 1101	ou	5D
0101 1110	ou	5E
0101 1111	ou	5F
0110 0000	ou	60

Il est prudent de spécifier par un symbole particulier (par exemple un h en indice) les nombres hexadécimaux. Ci-dessus, il ne faut pas confondre l'octet 60_h et le nombre décimal 60. De même, si une ambiguïté est possible, les nombres binaires seront indicés par un b.

Les octets servent au codage de toute l'information. Les variables sont codées par une suite d'octets, de même que les instructions des programmes. Nous ne nous intéresserons qu'au codage des variables. Il existe essentiellement deux types de variables :

- Celles qui sont codées par un nombre prédéfini d'octets. Le type de ces variables est dit simple. Il s'agit principalement des entiers, des nombres flottants, des booléens et des caractères.
- Celles qui sont codées par un nombre variable d'octets. Le type de ces variables est dit composé. Il s'agit des tableaux, des listes, des chaînes de caractères.

2- Les variables de type simple

a) Les entiers

Dans la plupart des langages de programmation, les entiers sont codés par un nombre prédéfini d'octets. La plupart des langages de programmation utilise un type dénommé **int16** d'entiers codés sur 2 octets, soit 16 bits, ou un type dénommé **int32** d'entiers codés sur 4 octets, soit 32 bits. Les représentations possibles des entiers sont donc en nombre fini. Dans le cas de **int32**, ils sont au nombre de 2^{32} et varient entre -2^{31} et $2^{31} - 1$. Cela convient à la plupart des applications ($2^{31} = 2147483648$), mais peut être insuffisant dans certains domaines mathématiques utilisant de grands nombres entiers.

L'avantage de ce type de variable est que les calculs sont parfaitement exacts tant qu'on reste dans le domaine de définition des entiers. Il n'y a pas d'erreurs d'arrondis. Les principales opérations qu'on utilise sur ce type de variables sont l'addition, la multiplication, la soustraction, l'élévation à une puissance entière et la division euclidienne (avec quotient entier et reste). Cette dernière est définie de la façon suivante. Si a et b appartiennent à $\mathbb{N}$, b étant non nul, il existe un unique couple (q, r) de $\mathbb{N}^2$ tel que :

$$a = bq + r \qquad \text{et} \qquad 0 \leq r < b$$

q est le quotient, r est le reste. Dans le cas de Scilab, la fonction modulo s'applique en fait à des variables de type réel.

Les opérations arithmétiques	
En Python	En Scilab
a+b, a*b, a-b, a**b q = a//b r = a%b	a+b, a*b, a-b, a**b q = a/b r = modulo(a,b)
En Maple	En Java

a+b, a*b, a-b, a^b
q := iquo(a,b);
r := irem(a,b);
	a+b, a*b, a-b, Math.pow(a,b)
	q = a/b;
	r = a%b;

L'inconvénient de ce type de variable est la limitation de sa capacité. Des calculs avec des nombres trop grands peuvent dépasser le domaine de définition des entiers.

Le codage des entiers **int32** se fait de la façon suivante. Si la suite de 32 bits est $a_{31}a_{30}a_{29}...a_2a_1a_0$, l'entier représenté vaut $\sum_{i=0}^{31} a_i\, 2^i$ modulo 2^{32}, ce qui signifie que :

Si $a_{31} = 0$, alors $0a_{30}a_{29}...a_1a_0$ représente l'entier $\sum_{i=0}^{31} a_i\, 2^i \in \{0, ..., 2^{31} - 1\}$

Si $a_{31} = 1$, alors $1a_{30}a_{29}...a_1a_0$ représente l'entier $\sum_{i=0}^{31} a_i\, 2^i - 2^{32} = \sum_{i=0}^{30} a_i\, 2^i - 2^{31} \in \{-2^{31}, ..., -1\}$

Les entiers forment alors un ensemble cyclique désigné en mathématiques par $\mathbb{Z}/2^{32}\mathbb{Z}$. On constate que le chiffre de gauche a_{31} désigne le signe de l'entier (+ si $a_{31} = 0$ et – si $a_{31} = 1$). On a ainsi :

suite de quatre octets sous forme hexadécimale	entier correspondant
00 00 00 00	0
00 00 00 01	1
00 00 00 02	2
00 00 00 03	3
...	...

00 00 00 09	9
00 00 00 0A	10
00 00 00 0B	11
00 00 00 0C	12
00 00 00 0D	13
00 00 00 0E	14
00 00 00 0F	15
00 00 00 10	16
00 00 00 11	17
...	...
7F FF FF FC	2147483644
7F FF FF FD	2147483645
7F FF FF FE	2147483646
7F FF FF FF	2147483647

On atteint ici le dernier entier positif ou nul ($2^{31} – 1$) codé en binaire 0111 1111 ... 1111. L'entier "suivant" est alors $– 2^{31}$, codé en binaire 1000 0000 ... 0000 :

80 00 00 00	–2147483648
80 00 00 01	–2147483647
80 00 00 02	–2147483646
...	...
FF FF FF FD	–3
FF FF FF FE	–2
FF FF FF FF	–1
00 00 00 00	0

EXEMPLE :

❑ Cette configuration permet de comprendre le comportement apparemment aberrant de certains logiciels. Ainsi, on souhaite calculer le plus petit commun multiple de 59356 et 44517. On a :

$a = 59356 = 14839 \times 4$

$b = 44517 = 14839 \times 3$

Le nombre $d = 14839$ est le plus grand diviseur commun de a et b. Le plus petit multiple commun est alors $m = 14839 \times 4 \times 3 = 178068$. Il existe des algorithmes efficaces du calcul du PGCD, basé sur le fait que PGCD(a, b) = PGCD(a modulo b, a), où a modulo b désigne le reste dans la division euclidienne de a par b. Une fois d calculé, on trouve le PPCM au moyen de la formule $m = \frac{ab}{d}$. Le logiciel Scilab (version 5.4) permet de calculer le PPCM au moyen de la fonction lcm appliquée sur les valeurs a et b converties au format **int32** :

```
lcm(int32([59356 44517]))
```

On obtient alors le résultat aberrant – 111369. Que s'est-il passé ? Si on demande le PGCD des deux valeurs (fonction gcd), on obtient bien $d = 14839$. Mais, au lieu de calculer le PPCM m en effectuant $(a/d) \times b$ qui donne le résultat exact sans difficulté, le logiciel calcule selon toute vraisemblance d'abord $a \times b$ entraînant un débordement de capacité. En effet, 59356×44517 vaut 2642351052 qui est supérieur à 2^{31}. Le logiciel le considère donc comme égal à $2642351052 – 2^{32} = – 1652616244$ dont le quotient entier par d vaut – 111369. On remarque donc que les deux formules, mathématiquement identiques, $m = (a/d) \times b$ et $m = (a \times b)/d$, ne le sont pas informatiquement. Le choix d'utiliser la deuxième formule est particulièrement regrettable. Il peut donner un résultat faux

même si le véritable PPCM est bien dans le domaine de définition des entiers, alors que la première formule aurait donné un résultat correct. On pourra ainsi tester que Scilab donne pour PPCM de 50000 avec lui-même la valeur surréaliste de – 35899. Ces problèmes ont été résolus à partir de la version 5.5 de Scilab.

En Python ou Maple, il n'y a pas de limite de capacité des entiers, ceux-ci étant codés par une suite arbitrairement longue d'octets lorsque la taille de quatre octets est dépassé. Cela se manifeste sous Python par l'affichage d'un L à l'écran apparaissant à la suite d'un tel entier long. Ces deux logiciels sont bien adaptés à la programmation de la suite de Collatz, vue plus haut, pour laquelle on ne sait pas prévoir jusqu'à quel maximum la suite va monter.

b) Les nombres flottants

Les nombres à virgule flottante ou nombres flottants ont pour but de représenter les nombres décimaux, avec une précision donnée. On distingue les flottants codés sur 32 bits ou 4 octets (flottants en simple précision ou **float32**) des flottants codés sur 64 bits ou 8 octets (flottants en double précision ou **float64**). Sous Python, le type **float** désigne le deuxième type. Ces derniers, plus précis, sont représentés en machine sous la forme $\pm\, m \times 2^n$ où le signe $\pm$ utilise 1 bit, m est un entier positif ou nul utilisant 52 bits et n une puissance entière relative utilisant les 11 bits restants. A l'affichage à l'écran, ce nombre est converti en notation scientifique sous la forme usuelle $\pm\, m' \times 10^{n'}$, où m' est un nombre décimal compris entre 1 et 10 et s'appelle la mantisse, n' étant l'exposant. Les 52 bits de m permettent de donner environ une quinzaine de chiffres significatifs exacts de m'. Les 11 bits de n permettent de faire varier n' entre – 300 et + 300 environ.

Dans la suite, pour simplifier, nous donnerons des exemples directement en décimal, sans oublier que les nombres introduits sont en fait convertis en binaire au sein de la machine pour exécuter les calculs, puis reconvertis en décimal pour affichage à l'écran. Voici des exemples de nombres flottants donnés avec 15 chiffres significatifs.

$a = 0.887987458369512 \times 10^5$

$b = -\,0.124445447874558 \times 10^{-3}$

Les nombres flottants ont une précision limitée ce qui signifie que, par essence, les calculs effectués avec eux sont approchés. Au cours des calculs, les erreurs peuvent d'ailleurs s'accumuler et entraîner un résultat infondé. Voyons ce qu'il en est par exemple pour l'addition de a et b ci-dessus. On aligne les chiffres des deux nombres sur le même exposant, mais il en résulte une perte de précision du nombre le plus petit :

0.887987458369512

– 0.000000001244454 7874558 tronqué à – 0.000000001244454

La somme $c = a + b$ vaut $0.887987457125058 \times 10^5$. Si l'on retranche a, on ne retrouve pas le b initial, mais le b tronqué.

$c - a = -\,0.000000001244454 \times 10^5 = -\,0.1244454 \times 10^{-3}$

Ainsi $(a + b) - a$ n'est pas numériquement égal à b.

De même, considérons les nombres suivants :

$a = 0.800000000002123$

$b = 0.800000000001526$

$c = 0.125478254 \times 10^{-10}$

Le calcul de $(a - b) + c$ donne :

$a - b = 0.597 \times 10^{-12}$

On remarquera qu'il n'y a que trois chiffres significatifs et qu'il ne saurait y en avoir plus, puisque les chiffres au-delà de 10^{-15} sont inconnus chez *a* et chez *b*.

$a - b + c = 0.131448254 \times 10^{-10}$

Le résultat donne apparemment neuf chiffres significatifs, mais les derniers chiffres 8254 sont dénués de signification puisque ceux correspondant à *a* et *b* sont inconnus. Effectuons maintenant le calcul dans l'ordre suivant :

$a + c = 0.800000000014671$

$a + c - b = 0.13145 \times 10^{-10}$

Il ne possède cette fois que cinq chiffres significatifs.

On prendra garde qu'un algorithme testant l'égalité de $a - b + c$ et de $a + c - b$ donnera un résultat généralement faux. Les tests d'égalité de nombres flottants sont à éviter. Si *x* est un nombre flottant, plutôt que de tester :

```
si x=0 alors ...
```

on préfèrera adopter un test du type :

```
si abs(x)<epsilon alors ...
```

où epsilon est une valeur choisie par le programmeur et qu'il estime sans risque pour confondre un *x* petit à un *x* nul. Cette question se pose par exemple pour la simple résolution d'une équation du second degré où il s'agit de tester si le discriminant est strictement négatif, nul ou strictement positif. Il n'existe pas de méthode totalement fiable de résolution d'une telle équation lorsqu'on se trouve au voisinage d'une racine double.

Les algorithmes de calcul sur nombres flottants peuvent être l'objet de propagation d'erreurs redoutables, et parfois difficiles à détecter. Il convient donc de ne jamais prendre un résultat numérique fourni par une machine au pied de la lettre, et de garder un regard critique sur ce résultat.

EXEMPLE 1 :

❑ On souhaite écrire une fonction de paramètre un entier *n* et donnant une valeur approchée de $I_n = \int_0^1 \frac{x^n}{10+x} dx$. On a $I_0 = \ln(\frac{11}{10})$ et, en écrivant que $x^n = x^{n-1}(10 + x) - 10x^{n-1}$, on obtient la relation $I_n = \frac{1}{n} - 10I_{n-1}$ d'où l'idée de calculer cette fonction par itération successive. Les logiciels de programmation possèdent des bibliothèques de fonctions permettant le calcul approché des fonctions usuelles (exp, ln, sin, cos, sqrt, etc...), ce qui permet d'initialiser I_0.

Calcul d'une intégrale	
En Python	En Scilab

`import numpy`
`def calculintegrale(n):`
`  I = numpy.log(11.0/10)`
`  for i in range(n):`
`  I = 1.0/(i+1) - 10*I`
`  return(I)`
	`function I = calculintegrale(n):`
	`  I = log(11/10)`
	`  for i = 1:n`
	`  I = 1/i - 10*I`
	`  end`

Le calcul des premières valeurs donne les résultats suivants (on n'en donne que 8 décimales) :

n	I_n
1	0.0468982
2	0.0310180
3	0.0231535
4	0.0184647
5	0.0153529
6	0.0131377
7	0.0114806
8	0.0101944
9	0.0091671
10	0.0083287
11	0.0076220
12	0.0071138
13	0.0057850
14	0.0135789
15	– 0.0691221
...	...
20	7483.468

La 14ème valeur est fausse. En effet, pour $x \in [0, 1]$, $\frac{x^n}{10 + x} \geq \frac{x^{n+1}}{x + 10}$ donc $I_n \geq I_{n+1}$ pour tout n. Or $I_{14} > I_{13}$. La valeur I_{15} est aberrante puisque négative. La 20ème l'est également puisque $0 \leq I_n \leq \int_0^1 x^n = \frac{1}{n + 1}$. La raison de ce dysfonctionnement provient du fait que, dès la première valeur, I_0 est approché à 10^{-15} près, mais que la récurrence multiplie l'erreur commise par 10 à chaque itération. L'erreur apparaît donc de manière flagrante au bout d'une quinzaine d'itérations.

Une méthode pour calculer de façon correcte I_n, n étant donné, consiste dans le cas présent à partir de I_{n+20} à qui on attribue l'approximation grossière $\frac{1}{n + 20}$, puis, pour k décroissant de $n + 19$ à n, à appliquer la relation de récurrence inverse $I_k = \frac{1}{10} (\frac{1}{k + 1} – I_{k+1})$. L'erreur initiale est divisée par 10 à chaque itération et deviendra inférieure à la précision des nombres flottants au bout de 20 itérations.

Calcul d'une intégrale	
En Python	En Scilab

def calculintegrale(n):
  I = 1.0/(n+20)
  for k in range(20):
  I = 1.0/10*(1.0/(n+20-k) - I)
  return(I)
	function I = calculintegrale(n)
	  I = 1/(n+20)
	  for k = 0:19
	  I = 1/10*(1/(n+20-k) - I)
	  end

Les premières valeurs donnent cette fois, de manière beaucoup plus fiable :

n	I_n
1	0.0468982
2	0.0310180
3	0.0231535

4	0.0184647
5	0.0153529
6	0.0131377
7	0.0114806
8	0.0101944
9	0.0091672
10	0.0083280
11	0.0076294
12	0.0070390
13	0.0065333
14	0.0060954
15	0.0057125
...	...
20	0.0043470

On a indiqué en bleu les chiffres qui diffèrent du premier calcul, ce qui fait apparaître la propagation des erreurs qui se sont produites dans le dit calcul.

EXEMPLE 2 :

❑ On souhaite écrire une fonction de paramètres *n* et qui calcule la quantité :

$$2^n \times \sqrt{2 - \sqrt{2 + \sqrt{2 + \sqrt{2 + ... + \sqrt{2 + \sqrt{2}}}}}}$$

constituée de *n* racines empilées. Il suffit de partir d'une variable valant initialement $\sqrt{2}$, puis de lui ajouter 2 et de prendre la racine carrée, et cela $n - 2$ fois. La dernière opération est la différence finale avec 2 avant de prendre une dernière racine carrée. On multiplie enfin par 2^n.

Calcul d'une racine	
En Python	En Scilab

import numpy as np
def f(n):
  s=np.sqrt(2)
  for i in range(n-2):
  s=np.sqrt(2+s)
  return(2**n*np.sqrt(2-s))
	function y = f(n)
	  s=sqrt(2)
	  for i=1:n-2
	  s=sqrt(2+s)
	  end
	  y=2^n*sqrt(2-s)
	endfunction

Voici les valeurs affichées par *f*(*n*) quand on incrémente *n* à partir de *n* = 2 :

3.06146745892
3.12144515226
3.13654849055
3.14033115695
3.14127725093
3.14151380114
3.14157294037
3.14158772528
3.1415914215
3.14159234561
3.14159257655

3.14159263346
3.14159265481
3.14159264532
3.14159260738
3.14159291094
3.1415941252
3.1415965537
3.1415965537
3.14167426502 bizarre, non ?
3.14182968189
3.14245127249
3.14245127249
3.16227766017
3.16227766017
3.46410161514
4.0 de plus en plus bizarre
0.0
0.0
0.0
0.0
0.0
0.0

Il est clair que le comportement numérique de la machine est suspect. La raison en est que, vers $n = 30$, la différence numérique entre $\sqrt{2+\sqrt{2+\sqrt{2+...+\sqrt{2+\sqrt{2}}}}}$ et 2 se situe au delà de la précision du calcul. Quand on fait la différence, on trouve un résultat numériquement nul, qui le restera même si on multiplie par le grand nombre 2^n. On peut néanmoins obtenir des valeurs approchées convenable de la suite en multipliant par la quantité conjuguée :

$$\sqrt{2-\sqrt{2+\sqrt{2+\sqrt{2+...+\sqrt{2+\sqrt{2}}}}}} = \frac{\sqrt{2^2-(2+\sqrt{2+\sqrt{2+\sqrt{2+...+\sqrt{2+\sqrt{2}}}}})}}{\sqrt{2+\sqrt{2+\sqrt{2+\sqrt{2+...+\sqrt{2+\sqrt{2}}}}}}}$$

$$= \frac{\sqrt{2-\sqrt{2+\sqrt{2+...+\sqrt{2+\sqrt{2}}}}}}{\sqrt{2+\sqrt{2+\sqrt{2+\sqrt{2+...+\sqrt{2+\sqrt{2}}}}}}}$$

Le nombre de racines du numérateur a diminué de 1 mais il est de la même forme que précédemment. On itère donc l'utilisation des conjugués pour arriver à l'expression finale suivante :

$$\frac{1}{\sqrt{2+\sqrt{2+\sqrt{2+\sqrt{2+...+\sqrt{2+\sqrt{2}}}}}}} \times \frac{1}{\sqrt{2+\sqrt{2+\sqrt{2+...+\sqrt{2+\sqrt{2}}}}}} \times ...$$

$$\times \frac{1}{\sqrt{2+\sqrt{2+\sqrt{2}}}} \times \frac{1}{\sqrt{2+\sqrt{2}}} \times \sqrt{2}$$

ce qui conduit à la fonction suivante, plus compliquée mais numériquement plus fiable :

Calcul d'une racine	
En Python	En Scilab

import numpy as np
def g(n):
p=np.sqrt(2)
r=np.sqrt(2+np.sqrt(2))
for i in range(n-1):
p=p/r
r=np.sqrt(2+r)
return(2**n*p)
	function y=g(n)
	p=sqrt(2)
	r=sqrt(2+sqrt(2))
	for i=1:n-1
	p=p/r
	r=sqrt(2+r)
	end
	y=2^n*p
	endfunction

On peut montrer en fait que l'expression qu'on cherche à calculer n'est autre que $2^{n+1}\sin(\frac{\pi}{2^{n+1}})$. On peut donc effectuer le calcul direct :

Calcul direct	
En Python	En Scilab
import numpy as np def h(n): return(2**(n+1)* np. sin(np.pi/2**(n+1)))	function y=h(n,x) y=2^(n+1)*sin(%pi/2^(n+1)) endfunction

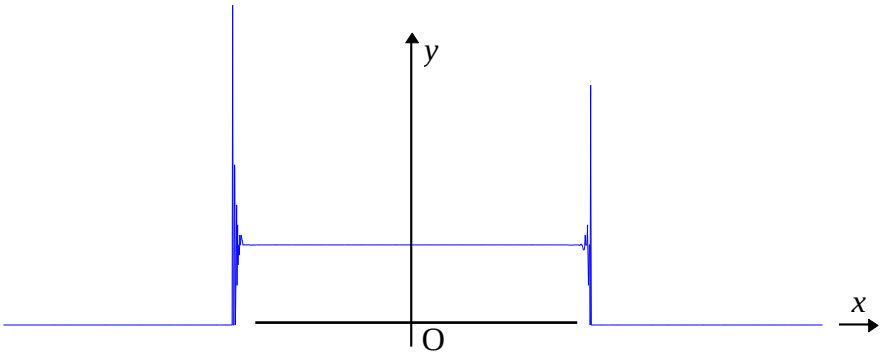
Même au-delà de $n = 30$, on pourra constater que $g(n)$ et $h(n)$ affichent la même valeur 3.14159265359, où l'on reconnaît une valeur approchée de π.

EXEMPLE 3 :

❑ On pose $\text{ch}(x) = \frac{e^x + e^{-x}}{2}$ et $\text{sh}(x) = \frac{e^x - e^{-x}}{2}$. Les logiciels mathématiques connaissent ces fonctions, généralement sous le nom cosh et sinh. Il en est de même des calculatrices scientifiques. On vérifiera facilement que, pour tout x, $\text{ch}^2(x) - \text{sh}^2(x) = 1$. Considérons la fonction :

$$f : x \to \text{ch}^2(x^2) - \text{sh}^2(x^2)$$

Cette fonction est mathématiquement constante égale à 1. Pourtant son tracé à l'aide d'un logiciel sur l'intervalle [–10, 10] donne :



Expliquer le phénomène.

Ainsi, la programmation d'une fonction peut sembler mathématiquement correcte, mais être numériquement problématique.

EXEMPLE 4 :
❑ Soit la suite (u_n) définie par $u_0 = 0.1$ et, pour tout n, $u_{n+1} = 3.8\, u_n(1 - u_n)$. Calculer u_{50} et u_{500}. Comparer avec le résultat obtenu en utilisant la définition $u_{n+1} = 3.8\, u_n - 3.8\, u_n^2$.

c) Les variables booléennes
Ce sont des variables qui ne peuvent prendre que deux valeurs, Vrai (True) et Faux (False). On utilise un octet pour les coder, 00_h pour Faux et 01_h pour Vrai. Elles sont utiles pour servir de drapeau signalant un état particulier d'un système. Par exemple, tout logiciel de traitement de données (traitement de texte, tableur, éditeur, ...) possède une variable booléenne prenant la valeur Vrai lorsqu'une sauvegarde des données est effectuée et prenant la valeur Faux dès qu'on modifie le document. Lorsque l'utilisateur souhaite fermer le document, le logiciel vérifie la nature de cette variable booléenne et, si elle vaut Faux, demande à l'utilisateur s'il souhaite sauvegarder les modifications.

Elles sont également utiles pour programmer une boucle d'itération du type tant que, lorsque la condition d'arrêt est complexe à formuler car pouvant provenir de plusieurs conditions :

```
Termine ← Faux
tant que non Termine faire
        <Blocs d'instruction dont certaines changent Termine en Vrai
        si une certaine condition d'arrêt est vérifiée>
finfaire
```

Il existe également des fonctions à valeurs booléennes. Ainsi, le logiciel Maple possède une fonction **isprime** définie sur les entiers et répondant Vrai ou Faux selon que le paramètre est un entier premier ou non.

d) Les caractères
Un caractère représente une lettre ou un chiffre et sont les éléments constitutifs des chaînes de caractères. Ils sont souvent codés sur un octet. Un octet permet de coder 256 caractères, suffisant pour traiter les majuscules et minuscules de l'alphabet latin. On peut également coder les caractères accentuées, mais ce codage n'est pas universel. De plus, le codage des divers alphabets du monde a conduit au développement de caractères au format Unicode pouvant être codés sur un nombre variable d'octets, allant parfois jusqu'à quatre octets voire plus.

e) Affectation de variables
La plupart des langages de programmation impose de déclarer le type de chaque variable avant son utilisation. D'autres définissent ce type implicitement au moment de la première affectation. La connaissance de ce type est nécessaire pour savoir quelles fonctions on peut appliquer à la variable. Par ailleurs, le type de la variable utilisée est nécessaire pour que le logiciel puisse réserver une place en mémoire adéquate à cette variable. Ainsi, lorsqu'on affecte une valeur de type entier long à une variable *a*, une adresse en mémoire est réservée à *a*. Cette adresse est la première de quatre octets successifs dans lesquels est stockée la valeur de *a*. Pour effectuer une opération avec *a*, il suffit d'aller lire à l'adresse affectée à *a* les données contenues à cette adresse.

On rencontre deux types d'affectation.

❑ Les affectations par valeurs se font de la façon suivante. Une affectation du type *b* ← *a* copie le contenu de l'adresse réservée à *a* à l'adresse réservée à *b*, et écrase donc tout ce qui était auparavant contenu à cette adresse. Les adresses de *a* et *b* sont différentes, mais contiennent la même valeur. Une modification ultérieure de *a* ne modifiera pas la valeur de *b*, sauf si on exécute à nouveau l'instruction *b* ← *a*.

```
a ← 3          # a prend la valeur 3
b ← a          # b prend la valeur 3
a ← 5          # a prend la valeur 5, mais b a toujours la valeur 3.
```

Ce type d'affectation est généralement utilisée pour les types simples (entier, flottant, booléen).

❑ Les affectations par adresses se font de la manière suivante. Une affectation *b* ← *a* attribue à *b* la même adresse que *a*. *a* et *b* sont alors deux noms synonymes pour désigner la même adresse. Une modification de *a* modifiera donc aussi *b*. Les affectations par adresses sont souvent effectuées pour les structures de données complexes et lourdes, et pour lesquelles la recopie des données demande un temps d'exécution non négligeable. Ces structures sont examinées au paragraphe suivant.

3- Les structures de données

On appelle structure de données un type de structure adoptée pour regrouper plusieurs variables. C'est le cas en particulier des tableaux et des listes. Dans les sous-paragraphes suivants a), b), c), d), nous ne ferons pas de différence autre que syntaxique entre ces deux structures, les considérant comme des structures indicées qui permettent de ranger une liste d'éléments les uns à la suite des autres. La syntaxe de déclaration et d'utilisation de telles structures est propre à chaque langage. Dans le sous-paragraphe e), nous préciserons davantage la différence théorique qu'on établit entre liste et tableau.

a) Les listes

Nous nous bornerons à donner des exemples de syntaxe sur les listes en Python. Les opérations usuelles qu'on mène sur les listes sont les suivantes :

❑ *Création*

```
a = [5,9,6]
```

L'instruction précédente crée une liste de 3 éléments. En Python, l'indice du premier élément est 0. Ci-dessus, l'indice du dernier élément est donc 2. On peut également créer un intervalle de valeurs entières successives au moyen de la fonction range(indicedeb, indicefin) ou range(indicedeb, indicefin, increment). Pour stocker dans une liste explicite les éléments de cet intervalle, on utilise à partir de la version 3 de Python la syntaxe list(range(indicedeb, indicefin)), l'indice de début étant inclus et l'indice de fin étant exclu. Cette commande est analogue en Scilab à indicedeb:indicefin-1. Si indicedeb vaut 0, on peut l'omettre.

❑ *Ajout et insertion d'un élément ou d'une sous-liste*

a[2:2]=[15,3,4]	insère des éléments au sein d'une liste, à partir de l'indice 2
a.insert(5,12)	insére en indice 5 de a l'élément de valeur 12
a[2:7]=[15,3,4]	remplace les éléments d'indice 2 à 6 par la liste [15,3,4]
a.append(18)	ajoute l'élément 18 en fin de liste
a.extend (b)	concatène la liste *b* en fin de liste *a*
c=a+b	crée une liste *c* en concatènant les deux listes *a* et *b*

❑ *Suppression*

del a[1:4]	supprime les éléments d'indice 1 à 3
a.pop()	enlève le dernier élément de la liste,

❑ *Accès à un élément*

a[5]	5ème élément de la liste *a*
a[5]=42	changement de la valeur du 5ème élément de la liste
a[5:9]	sous-liste depuis l'indice 5 inclus, à l'indice 9 exclu

❑ *Fonctions diverses*

len(a)	longueur d'une liste

b) Les tableaux

Les langages de programmation possèdent également des structures indicées appelées tableaux. Ceux-ci sont adaptés aux problèmes où le nombre d'éléments est connu à l'avance, correspondant par exemple aux notions de vecteurs ou de matrice en mathématiques.

Voici des exemples de tableaux en Python, utilisant la classe ndarray du module numpy. La première instruction se borne à une déclaration d'un tableau T constitué de 10 éléments sans les définir explicitement pour le moment, la seconde définit un tableau U par ses six éléments, la troisième définit un tableau V de longueur 15 uniquement constitué de 1, la quatrième définit un tableau W à deux indices, pouvant servir à représenter une matrice 2×4.

```
import numpy
T = numpy.empty(10)
U = numpy.array([1,2,3,5,8,13])
V = numpy.ones(15)
W = numpy.array([ [1,-2,3,4] , [5,-3,4,0] ])
```

En Scilab, on écrira :

```
U = [1,2,3,5,8,13]
V = ones(15)
W = [ [1,-2,3,4] , [5,-3,4,0] ]
```

On accède au *i*-ème élément du tableau U par la syntaxe U[i] en Python et U(i) en Scilab. La syntaxe de Scilab est peu heureuse car elle ne distingue pas l'élément d'un tableau du calcul d'une fonction appliqué sur un élément. Dans la suite, nous utiliserons donc les crochets, plus répandus parmi les langages de programmation. Par ailleurs, par défaut, le premier indice du tableau est 0 en Python ou Java, mais 1 en Scilab ou Maple, ce qui peut entraîner des désagréments quand on passe d'un langage à l'autre. Ainsi, avec les exemples précédents, U[0] vaut 1 en Python, et U[1] vaut 2, alors que ces deux éléments sont respectivement en Scilab par U(1) et U(2). En ce qui concerne le tableau W, on utilise la syntaxe W[i,j] ou W[i][j] ou W(i,j) selon les langages. Une demande d'accès à un élément du tableau pour un indice au-delà de la longueur du tableau entraîne une erreur d'exécution. Il existe des instructions (size ou shape) spécifique à chaque langage pour connaître la longueur d'un tableau.

c) Exemples

Soit T un tableau ou une liste possédant *n* éléments, de T[0] jusqu'à T[*n*–1]. Voici quelques algorithmes relatifs à ce tableau.

Recherche d'un élément particulier *k* dans le tableau

On se donne un élément *k* que l'on cherche dans le tableau T. Si *k* s'y trouve, l'algorithme s'arrête au premier *i* tel que T[*i*] = *k* et donne la valeur de ce *i*. Si *k* n'existe pas, la fonction donnera –1 comme résultat.

```
i ← 0
tant que T[i] <> k et i<n-1 faire i ← i+1 finfaire    # on s'arrête si T[i] = k ou si i = n-1 indice
                                                      # du dernier élément du tableau
si T[i] = k alors i sinon -1 finsi
```

L'inégalité $i < n - 1$ doit être stricte car, si on avait $i \leq n - 1$ et *k* absent du tableau, alors *i* prendrait nécessairement la valeur *n* en fin de boucle et le test T[*i*] <> *k* entraînerait un débordement du tableau et une erreur d'exécution.

Recherche du plus petit élément d'un tableau de nombres flottants

m désignera ce plus petit nombre et *j* son indice. On parcourt évidemment la totalité du tableau (itération *pour* contrairement à l'exemple précédent qui utilisait une itération *tant que*) :

```
m ← T[0]                          # m est le minimum du tableau entre les indices 0 et i
j ← 0                             # j est l'indice de ce minimum.
pour i de 1 à n-1 faire
    si T[i] < m alors m ← T[i]
                      j ← i finsi
finfaire
```

Calcul de la moyenne d'un tableau de nombres flottants

Une variable auxiliaire S accumule la somme des termes les uns après les autres. Arrivé en fin de tableau, il suffit de la diviser par le nombre d'éléments.

```
S ← 0
pour i de 0 à n-1 faire S ← S + T[i] finfaire
moyenne := S/n
```

Calcul de la variance d'un tableau de nombres flottants

La variance est définie comme étant égale à $\frac{1}{n}\sum_{i=0}^{n-1}(\mathrm{T}[i] - m)^2$ où $m = \frac{1}{n}\sum_{i=0}^{n-1}\mathrm{T}[i]$ est la moyenne. En développant le carré, elle vaut également $\frac{1}{n}\sum_{i=0}^{n-1}\mathrm{T}[i]^2 - m^2$. Si la moyenne a déjà été calculée, on peut opérer comme suit :

```
V ← 0
pour i de 0 à n-1 faire
        V ← V + (T[i] - moyenne)**2      # ** est l'élévation à une puissance
finfaire
variance := V/n
```

Sinon, on peut calculer moyenne et variance en une seule boucle :

```
S ← 0
```

```
V ← 0
pour i de 0 à n-1 faire
        S ← S + T[i]
        V ← V + T[i]**T[i]
finfaire
moyenne := S/n
variance = V/n - m**2
```

Cependant cette deuxième méthode amène souvent à ajouter dans le calcul de V des termes $T[i]^2$ dont l'ordre de grandeur peut varier notablement, ce qui entraîne une imprécision sur V, une deuxième imprécision étant due au calcul final de la variance qui retranche deux nombres qui peuvent être grands, mais comparables. On retrouve ici les difficultés évoquées sur le calcul avec les nombres flottants.

d) Affectation de variables

Soit *a* une liste ou un tableau. Le logiciel attribue à *a* une adresse à partir de laquelle il est capable de trouver les éléments de *a*. Que fait l'instruction suivante ?

```
b ← a                    # b = a en Scilab ou Python
```

Il y a deux interprétations possibles radicalement différentes.

- Ou bien le logiciel crée un nouveau tableau ou liste *b* et il copie dans la liste *b* la liste des éléments de *a*. *b* est alors un clone de *a*, les deux étant situés à des adresses physiquement différentes. Une modification ultérieure de *a* ne modifiera pas *b*. Cette méthode est cependant encombrante et lente si la structure est constituée de nombreuses données.
- Ou bien le logiciel attribue à *b* la même adresse que *a*. L'attribution est alors très rapide, mais *a* et *b* sont alors deux noms différents pour le même objet et toute modification de l'un entraîne une modification de l'autre.

Ainsi, en Scilab :

```
A = [1 2 3]
B = A                    // B est une copie de A
B(2) = 5                 // B vaut maintenant [1 5 3], mais A n'a pas changé
```

alors qu'en Python :

```
a = [1,2,3]
b = a                    # b se voit affecté la même adresse que a. a et b sont
                         # deux noms du même objet
b[1] = 5                 # b vaut maintenant [1,5,3], mais a aussi !!
c = a[:]                 # Ici, on copie dans c les éléments de a. c est un clone de a
                         # physiquement distinct de celui-ci.
c[1] = 6                 # c vaut maintenant [1,6,3] mais a (ni b) n'a pas changé
```

Il convient donc de bien comprendre si l'instruction ← copie une valeur ou copie une adresse. Le comportement des variables est totalement différent dans les deux cas.

e) Temps de calcul

Il convient de mener une réflexion sur le temps de calcul lié à l'utilisation des listes ou tableaux. Ce temps de calcul est directement lié à la façon dont ces structures sont implémentées dans la mémoire de la machine. En principe, les noms de tableau ou de liste désignent deux modes différents d'implémentation de ces structures.

❑ **Les tableaux :** la machine réserve en mémoire une place successive à chaque élément du tableau a constituée de *n* éléments. Dans ce cas, connaissant l'indice *i* d'un élément, la machine est capable de déterminer à quelle adresse précise se trouve l'élément a[i]. L'accés à cet élément se fait alors en un temps indépendant de *n*. On dira que le temps d'accès est en O(1). Par contre l'insertion d'un nouvel élément ou la suppression d'un élément à un indice *i* donné demande de décaler tous les éléments à la droite de cet indice, vers la droite en cas d'insertion pour laisser de la place au nouvel élément, vers la gauche en cas de suppression pour éliminer la place laissée vacante. Le temps d'exécution d'une suppression ou d'une insertion est alors en O(*n*). Ce type de structure est particulièrement adapté lorsque des accès aux éléments de la liste sont nombreux, mais qu'il y a peu d'insertion ou de suppression d'éléments.

❑ **Les listes :** la machine stocke en mémoire les éléments de façon chaînée. Partant d'un élément appelé tête de la liste, chaque élément a connaissance de l'adresse où se trouve l'élément suivant. Pour accèder au *i*-ème élément, on parcourt la chaîne depuis la tête jusqu'à l'élément désiré et le temps d'accès au *i*-ème élément est en O(*n*). Par contre, une fois atteint cet élément *i*, la suppression de l'élément $i + 1$ se fait en O(1) puisqu'il suffira d'indiquer à l'élément *i* quelle est l'adresse de l'élément $i + 2$. De même, une fois atteint l'élément *i*, l'insertion d'un nouvel élément entre celui-ci est le suivant se fait en O(1) : on indique à l'élément *i* l'adresse du nouvel élément, et on indique au nouvel élément l'adresse de l'élément situé auparavant en $i + 1$. Ce type de structure est particulièrement adapté aux données où l'on fait de nombreuses insertions ou suppressions à partir d'un point donné (c'est typiquement le cas des traitements de textes par exemple). L'avantage de cette structure réside également dans le fait qu'on n'a pas besoin de réserver a priori en mémoire une plage d'adresses successives pour stocker les données. Celles-ci peuvent être dispersées. C'est intéressant quand on ignore a priori quelle sera la taille finale de la liste.

Selon les cas, c'est à l'utilisateur de déterminer quelle est le type de structure le mieux adapté au problème qu'il se pose et à consulter la documentation pour savoir si le langage de programmation qu'il utilise prévoit le type de données qu'il souhaite et sa forme syntaxique. La différence n'est pas anodine lorsque *n* s'élève à plusieurs centaines de milliers de données.

f) Les chaînes de caractères

Ce type de variable (**string**) permet de représenter des mots ou même des phrases. Chaque langage possède des fonctions pour traiter ce genre de variable (longueur, recherche d'un sous-mot, insertion, suppression, ...). On peut la voir comme une liste de caractères.

Voici un exemple de définition de mot en Python :

```
mot = "bonjour"
```

On peut accéder à la *i*-ème lettre du mot au moyen de `mot[i]`, la première lettre ayant pour indice 0. La longueur d'un mot est donné par `len(mot)`. On peut concaténer deux mots l'un après l'autre à l'aide de l'opérateur + :

```
mot1 = "abra"
mot2 = "cadabra"
print(mot1+mot2)
```

On peut extraire un sous-mot d'un mot en indiquant l'indice de la première lettre du sous-mot (inclus) et l'indice de la dernière lettre (exclue) : `mot[1:4]`. Pour d'autres langages de programmation, les conventions précédentes (en particulier sur les indices) peuvent être différentes. Ainsi, en Scilab, l'indice des lettres commence à 1.

Voici par exemple comment on peut chercher un sous-mot de *p* lettres dans un mot de *n* lettres, en ne s'autorisant qu'à accéder aux lettres de chaque mot. *i* étant donné entre 0 et *n* – *p*, on compare pour *j* variant de 0 à *p* – 1 les lettres sousmot[i+j] à mot[j]. Si les *p* lettres coïcident, le sous-mot est bien inclus dans le mot à partir du rang *i*. Sinon, on augmente *i* de 1. La boucle sur *j* n'a pas besoin d'être menée à terme si une lettre diffère. Une boucle tant que s'impose donc. De même la boucle sur *i* s'arrête dès que le sous-mot a été trouvé. Il est commode d'utiliser une variable booléenne Trouve servant de drapeau ; elle prend la valeur Vrai tant qu'on n'a pas trouvé de lettre qui diffère entre le sous-mot et le mot. Si cette variable a gardé la valeur vrai lorsqu'on a passé en revue le sous-mot en entier, c'est que ce dernier est bien inclus dans le mot. Initialement, on donne à Trouve la valeur Faux afin de démarrer la boucle sur *i*. On a mis en commentaire les relations vérifiées par les variables au cours du calcul. Dans ces commentaires, on note mot[i...k] le sous-mot constitué des lettres *i* à *k* incluses du mot (plutôt que mot[i:k+1] comme plus haut). On donne comme résultat le couple constitué de la variable booléenne Trouve et de l'indice à partir duquel se trouve le sous-mot, ou –1 si le sous-mot n'est pas inclus dans le mot. On suppose qu'on accède à la lettre *i* du mot à l'aide de la syntaxe mot[i].

```
Trouve ← Faux
i ← 0
tant que (non Trouve) et (i<=n-p) faire
        Trouve ← Vrai
        j ← 0                                   # sousmot[0...j-1] = mot[i...i+j-1] = mot vide
        tant que Trouve et (j<p) faire          # sousmot[0...j-1] = mot[i...i+j-1] est un invariant
                                                # de boucle
                si sousmot[j]<>mot[i+j] alors   # sousmot n'est pas inclus dans mot
                        Trouve ← Faux
                sinon                           # sousmot[0...j] = mot[i...i+j]
                        j ← j+1                 # sousmot[0...j-1] = mot[i...i+j-1]
                finsi   # ou bien Trouve = Faux ou bien sousmot[0...j-1] = mot[i...i+j-1]
        finfaire        # ou bien Trouve = Faux ou bien (j=p et sousmot[0...p-1] = mot[i...i+p-1])
                        # et donc : ou bien Trouve = Faux ou bien sousmot est inclus dans mot
                        # à partir du rang i
        si non Trouve alors i ← i + 1
finfaire                # ou bien Trouve = Vrai (et donc sousmot est inclus dans mot dès le rang i)
                        # ou bien i = n-p+1 et sousmot n'est pas inclus dans mot
si Trouve alors resultat ← [Trouve, i]
sinon resultat ← [Trouve,-1]
```

Recherche d'un sous-mot	
En Python	En Scilab

def mongrep(sousmot,mot):
  n = len(mot)
  p = len(sousmot)
  Trouve = False
  i = 0
  while (not Trouve) and (i<=n-p):
  Trouve = True
  j = 0
  while Trouve and (j<p):
  if sousmot[j] != mot[i+j]:
  Trouve = False
  else:
  j = j+1
  if not Trouve:
  i = i + 1
  if Trouve:
  return([Trouve, i])
  else:
  return([Trouve,-1])
	function y = mongrep(mot,sousmot)
	  n = length(mot)
	  p = length(sousmot)
	  Trouve = %F
	  i = 1
	  while (~Trouve) & (i<=n-p+1)
	  Trouve = %T
	  j = 1
	  while Trouve & (j<=p)
	  if part(sousmot,j)<>part(mot,i+j-1) then
	  Trouve = %F
	  else j = j+1
	  end
	  end
	  if ~Trouve then i = i + 1
	  end
	  end
	  if Trouve then y = list(Trouve, i)
	  else y = list(Trouve,-1)
	  end
	endfunction

III : Questions diverses relatives aux algorithmes

1- L'équation du second degré et la résolution d'équation par dichotomie

Nous avons évoqué en parlant des nombres flottants la difficulté à résoudre une équation du second degré. Considérons par exemple une équation $x^2 + bx + c = 0$, b et c réels, de discriminant $d = b^2 - 4c$ et considérons l'algorithme suivant, supposé donner le nombre nbs de solutions réelles :

```
d ← b**2 - 4*c
si d>0
   alors nbs ← 2
   sinon
      si d=0
         alors nbs ← 1
         sinon nbs ← 0
       finsi
   finsi
```

Tout est mathématiquement correct, mais informatiquement, le test d=0 pose numériquement un problème. En effet, on ne connait qu'une valeur approchée de d, et, si d est trop proche de 0, la machine lui donnera une valeur approchée dont il est impossible de savoir a priori si elle sera strictement positive, nulle, ou strictement négative. Le résultat de l'algorithme sera alors incertain et pourra dépendre de la machine utilisée et du langage de programmation utilisé.

Il n'existe aucun algorithme répondant à la question posée de façon certaine. En effet, un tel algorithme, s'il existait, signifierait qu'à l'issue d'un calcul, on est capable de dire si un nombre est

rigoureusement nul. Or il n'existe aucun procédé général qui puisse répondre à cette question. A titre d'exemple, comment savoir si le nombre suivant est nul ?

$$d = \sqrt{5} + \sqrt{22 + 2\sqrt{5}} - \sqrt{11 + 2\sqrt{29}} - \sqrt{16 - 2\sqrt{29} + 2\sqrt{55 - 10\sqrt{29}}}$$

Quand on demande une valeur approchée de d à Xcas, il donne une valeur négative, quand on demande à Python, geogebra ou à un tableur, la réponse est nulle, quand on demande à Maple, la réponse est positive.

Si le cas précédent paraît trop simple au lecteur, considérer la suite $(d_n)_{n\geq 1}$ définie par :

$d_n = 0$ si $n^{17} + 9$ et $(n + 1)^{17} + 9$ admettent 1 comme seul diviseur commun

$d_n = 1$ si n est pair et $n^{17} + 9$ et $(n + 1)^{17} + 9$ ont un autre diviseur commun que 1

$d_n = -1$ si n est impair et $n^{17} + 9$ et $(n + 1)^{17} + 9$ ont un autre diviseur commun que 1

et soit $d = \sum_{n=1}^{\infty} \frac{d_n}{10^n}$. d est-il nul ? positif ? négatif ? On remarquera que, dans les deux exemples donnés, on peut calculer d à n'importe quelle précision désirée.

Concernant les équations du second degré, le seul algorithme raisonnable est celui qui donne des valeurs approchées des solutions **complexes**. En effet, même si par exemple d est numériquement négatif pour la machine alors qu'il est mathématiquement positif ou nul, les solutions complexes approchées auront certes des parties imaginaires mais celles-ci seront très faibles, et les solutions complexes données seront numériquement très proches des solutions réelles.

Le même problème se pose pour la résolution d'une équation par dichotomie. Le problème de concours CPGE E3A 2017 de la filière PSI demandait d'écrire en Python une fonction `rech_dicho(f,a,b,eps)` de paramètre une fonction f continue, deux réels a et b tels que f soit définie sur $[a, b]$ avec $f(a)f(b) < 0$ et une marge d'erreur `eps`, cette fonction devant donner une valeur approchée d'une racine r de f à `eps` près en procédant par dichotomie (clairement inspiré d'une démonstration par dichotomie du théorème des valeurs intermédiaires). Un tel algorithme n'existe malheureusement pas. Considérons en effet un nombre d élément de $]-1, 1[$ et soit f la fonction suivante définie sur $[0, 1]$:

$$f(x) = (3d + 3)x - 1 \quad \text{si } x \leq \frac{1}{3}$$

$$= d \quad \text{si } \frac{1}{3} \leq x \leq \frac{2}{3}$$

$$= (3 - 3d)x + 3d - 2 \quad \text{si } x \geq \frac{2}{3}$$

On vérifiera facilement que, mathématiquement :

cette fonction f est continue et croissante sur $[0, 1]$

si $d > 0$, la seule racine de f est $\frac{1}{3 + 3d} < \frac{1}{3} < \frac{1}{2} - \frac{1}{10}$

si $d = 0$, les racines de f forme l'ensemble $[\frac{1}{3}, \frac{2}{3}]$

si $d < 0$, la seule racine de f est $\frac{2 - 3d}{3 - 3d} > \frac{2}{3} > \frac{1}{2} + \frac{1}{10}$

L'algorithme `rech_dicho` prétend trouver une valeur approchée d'une racine de f. Appliquons alors le `rech_dicho` supposé exister à cette fonction, en prenant pour d par exemple la valeur $\sqrt{5} + \sqrt{22 + 2\sqrt{5}} - \sqrt{11 + 2\sqrt{29}} - \sqrt{16 - 2\sqrt{29} + 2\sqrt{55 - 10\sqrt{29}}}$, $a = 0$, $b = 1$, `eps` $= \frac{1}{10}$ (nous

supposons que le prétendu algorithme a la possibilité de calculer une valeur approchée de d à la précision qu'il souhaite). Si la réponse est comprise entre 0.4 et 0.6, on serait alors certain que, mathématiquement $d = 0$. Ainsi, l'algorithme par dichotomie demandé par le concours serait capable dans ce cas de prouver l'égalité algébrique :

$$\sqrt{5} + \sqrt{22 + 2\sqrt{5}} - \sqrt{11 + 2\sqrt{29}} - \sqrt{16 - 2\sqrt{29} + 2\sqrt{55 - 10\sqrt{29}}} = 0$$

Mieux, si on l'applique au deuxième $d = \sum_{n=1}^{\infty} \frac{d_n}{10^n}$ proposé plus haut (il suffit pour cela d'intégrer dans la fonction définissant f un calcul approché de d à toute précision demandée), et si la réponse de `rech_dicho` était encore comprise entre 0.4 et 0.6, cela signifierait que l'algorithme est capable de dire que pour tout n, le pgcd de $n^{17} + 9$ et $(n + 1)^{17} + 9$ vaut 1.
Un autre réponse de `rech_dicho` conduirait à d'autres résultats invraisemblables. Il est inconcevable que, de l'algorithme demandé, on puisse en déduire une égalité algébrique ou l'existence d'un entier vérifiant une propriété arithmétique donnée.

Il eut été plus raisonnable que l'algorithme par dichotomie `rech_dicho(f,a,b,eps)` se borne à demander une valeur x telle que $|x - r| <$ `eps` (r racine de f) ou que $|f(x)| <$ `eps`. Autrement dit, on demande une valeur x qui approche la racine à `eps` près, ou alors une valeur x telle que $f(x)$ soit nulle à `eps` près.

```
def rech_dicho(f,a,b,eps):
    mini=a
    maxi=b
    c=(mini+maxi)/2
    tant que (maxi-mini>eps) et abs(f(c))>eps
        si f(mini)*f(c)<0
            maxi=c
        sinon
            mini=c
        finsi
        c=(mini+maxi)/2
    fintantque
    return(c)
```

2- Preuve d'un algorithme

Le fait d'écrire un programme et de le tester sur quelques exemples pour voir s'il marche ne permet pas d'assurer la validité de l'algorithme utilisé. Pour cela, il faudrait tester toutes les données possibles, ce qui est en général impossible, les données décrivant en général un ensemble infini. Le seul résultat que peut apporter un test est le suivant : si le résultat fourni par la machine n'est pas celui prévu, alors il est certain que l'algorithme est erroné. L'utilisation des tests est donc non pas de valider un algorithme, mais de le réfuter. On peut faire un rapprochement avec :

- les mathématiques : il faut une démonstration pour prouver un théorème. Des exemples, aussi nombreux soient-ils ne suffisent pas à le valider. Par contre, un seul contre-exemple suffit à le réfuter.
- la physique : une théorie physique permet de prévoir le résultat de certaines expériences, sous certaines hypothèses. Une théorie n'est jamais assurée. Sa validité peut n'être que temporaire, jusqu'à ce que soit réalisé une expérience qui la réfute. Ainsi, l'expérience négative de Michelson et Morley, en 1887, destiné à "prouver" le mouvement relatif de la Terre par rapport à un milieu hypothétique –

l'éther – n'a eu qu'une conséquence, celui de réfuter l'hypothèse de l'existence de l'éther (au grand désappointement des deux physiciens, d'ailleurs).

Un algorithme se prouve au même titre qu'un théorème mathématique. La plupart des algorithmes que nous avons donnés ont leur preuve indiquée en bleu en commentaire. Il s'agit essentiellement de prouver qu'une itération produit bien ce pour quoi elle a été conçue. Le raisonnement consiste à mettre en évidence un invariant de boucle, propriété qui est vraie avant la première itération, qu'on suppose vraie au début de la *i*-ème itération et dont on vérifie qu'elle est encore vraie à l'issue de la *i*-ème itération. Elle sera donc vraie à la sortie de boucle et on vérifie alors si la valeur de l'invariant de boucle à la sortie de l'itération donne bien le résultat attendu.

Outre les exemples déjà donnés, en voici un dernier. Considérons l'algorithme suivant dont le paramètre est une variable X donnée par l'utilisateur, appartenant à l'intervalle [1, 10[. E est un nombre petit servant de marge d'erreur (par exemple 10^{-10}).

```
S ← 1
Y ← 0
Z ← X
tant que S > E faire
        S ← S/2
        si Z^2 >= 10 alors Z ← Z^2/10
                           Y ← Y+S
                    sinon Z ← Z^2
        finsi
finfaire:
```

Cet algorithme calcule le logarithme décimal du nombre X. Si Y est ce logarithme, on a $X = 10^Y$. En fait, on ne calcule qu'une valeur approchée de Y à l'erreur E près. Voici la preuve de cette affirmation. Les variables utilisées sont les suivantes :

X	: variable dont on cherche le logarithme.
Y	: valeur approchée du logarithme
E	: précision à laquelle est calculée le logarithme
Z	: reste intervenant dans la différence entre Y et la valeur exacte de $\log_{10}(X)$.
S	: variable indiquant la précision du calcul effectué.

Reprenons les instructions en ajoutant des commentaires :

```
S ← 1
Y ← 0
Z ← X
# Initialement, on a log10(X) = Y + S*log10(Z), avec 1 ≤ Z < 10. Cette relation
# est l'invariant de boucle, ce qu'il convient de vérifier :
tant que  S ≥ E faire
        S ← S/2
        # Puisque S est divisé par 2, on a maintenant log10(X) = Y + 2S*log10(Z)
        # ou encore log10(X) = Y + S*log10(Z^2),
        # et aussi log10(X) = Y + S + S*log10(Z^2/10) compte tenu du fait que log10(10) = 1
        si Z^2 >= 10 alors Z ← Z^2/10
                           # Si Z^2 ≥ 10, on se ramène à un nombre Z entre 1 et 10.
                           # On a maintenant log10(X) = Y + S + S*log10(Z) et 1 ≤ Z < 10
                           Y ← Y+S
                           # On a maintenant log10(X) = Y + S*log10(Z) et 1 ≤ Z < 10
                    sinon Z ← Z^2
```

```
                    # Comme on avait log10(X) = Y + S*log10(Z²) avant le "si",
                    # on a maintenant log10(X) = Y + S*log10(Z) et 1 ≤ Z < 10
        finsi
        # Dans les deux cas, on a à nouveau log10(X) = Y + S*log10(Z), avec 1 ≤ Z < 10
        # L'invariant de boucle est bien validé
finfaire
# Quand on sort de la boucle, on a log10(X) = Y + S*log10(Z) et 1 ≤ Z < 10 et S < E
# donc Y ≤ log10(X) < Y + E. Y donne bien une valeur de log10(X) à moins de E près
# Il vaut mieux d'ailleurs prendre comme valeur approchée Y + E/2.
```

Il convient d'apporter une autre preuve, celle que le programme se termine. En effet, si l'utilisation d'une boucle pour un indice *i* variant de ... à ... est nécessairement finie, il n'en est pas de même des boucles tant que dont on ignore si elle ne pourrait pas boucler indéfiniment. La preuve repose ici sur le fait que la valeur de S est divisée par 2 à chaque itération, donc deviendra inférieure à E strictement positif.

3- Complexité d'un algorithme

Considérons deux algorithmes résolvant le même problème. Comment savoir quel est le plus rapide ? Le fait de les essayer l'un et l'autre sur les mêmes données ne suffit pas, car le résultat peut dépendre de la machine utilisée ou des données, et d'autre part, il est impossible de tester toutes les données possibles si elles sont en nombre infini, ce qui est généralement le cas. On associe donc à chaque donnée un nombre n directement lié à la complexité du problème. Ainsi, pour le tri du tableau, n peut être la dimension du tableau. Pour chaque algorithme, on évalue le nombre d'instructions effectuées en fonction de n. Seul l'ordre de grandeur de cette quantité nous intéresse. On se contente en général d'évaluer le nombre d'itérations réalisées.

EXEMPLES :

❑ Calcul de b^a en fonction de l'entier a : Si on multiplie b par lui-même a fois, le temps de calcul est en $O(a)$. Cependant, l'algorithme de l'exponentiation rapide donné en fin du I) divise a par 2 (division entière) jusqu'à ce qu'il s'annule. Si n est tel que $2^{n-1} \leq a < 2^n$, alors l'algorithme se termine en n itérations, et le temps de calcul est en $O(\ln(a))$, négligeable devant $O(a)$.

❑ La recherche du plus petit élément d'un tableau numérique : L'algorithme donné plus haut consiste à parcourir l'ensemble du tableau. Le temps de calcul est en $O(n)$. Mais si on sait que le tableau est trié par ordre croissant, il suffit évidemment de prendre le premier élément du tableau. Le temps de calcul est en $O(1)$ dans ce dernier cas.

❑ La recherche d'un élément donné d'un tableau numérique : L'algorithme donné plus haut consiste également à parcourir l'ensemble du tableau. Le temps de calcul est en $O(n)$. Mais si on sait que le tableau est trié par ordre croissant, on peut procéder par dichotomie. On considère un élément situé au centre du tableau et si l'élément cherché est plus grand, on poursuit itérativement la recherche dans la partie droite du tableau, sinon on poursuit dans la partie gauche. Ces deux parties sont constituées de $\frac{n}{2}$ éléments. La dichotomie revient donc à chaque fois à chercher l'élément dans un sous-tableau deux fois plus petit que le tableau précédent. Comme pour l'exponentiation rapide, le temps de calcul est en $O(\ln(n))$ négligeable devant n. Si de nombreuses recherches doivent être faites, il convient de décider s'il ne faut pas une fois pour toute trier le tableau. Les algorithmes de tri sont exposés dans L2/ALGO2.PDF.

❑ La recherche d'un sous-mot de longueur p dans un mot de longueur n. L'algorithme donné consiste dans le pire des cas à chercher pour tout indice i, variant de 0 à $n - p$, si les p lettres du mot à partir de cet indice coïncide avec les p lettres du sous-mots. Pour chaque i, il faut p itérations pour comparer le sous-mot aux lettres du mot. Le temps total de calcul est un O(np).

❑ **L'algorithme de Horner** : Soit a un scalaire et $P = \sum_{k=0}^{n} \lambda_k X^k$ un polynôme. On souhaite évaluer le nombre de produits qu'il est nécessaire d'effectuer pour calculer P(a), en fonction du degré n du polynôme. Si on calcule $\lambda_k a^k$ en multipliant λ_k par a k fois de suite (soit k produits), on aura besoin de $\sum_{k=0}^{n} k = \frac{n(n+1)}{2} = O(n^2)$ produits.

Si on calcule a^k par exponentiation rapide, il existe une constante C telle que, pour tout k, le nombre de produits nécessaires pour le calcul de $\lambda_k a^k$ soit majoré par C$\ln(k)$. Le nombre total de produits est alors majoré par :

$$\sum_{k=1}^{n} C\ln(k) \le \sum_{k=1}^{n} C \int_{k}^{k+1} \ln(x)\,dx \le C \int_{1}^{n+1} \ln(x)\,dx$$
$$\le C\,[x\ln(x) - x]_{1}^{n+1} = O(n\ln(n))$$

avec O($n\ln(n)$) négligeable devant O(n^2) quand n tend vers l'infini.
La méthode de Horner, quant à elle, consiste à écrire que :

$$P(a) = ((...((\lambda_n a + \lambda_{n-1})a + \lambda_{n-2})a + ...)a + \lambda_1)a + \lambda_0$$

et le nombre de produits est n, négligeable devant O($n\ln(n)$). L'algorithme est le suivant, en notant p la variable dont la valeur finale sera P(a), et L[k] les coefficients λ_k, supposés stockés dans un tableau L :

```
p ← L[n]                                   # valeur initiale de p
Pour i décroissant de n-1 à 0 faire
# Au début de la boucle, i = n et  p = L[n]a^(n–i–1) + L[n-1]a^(n–i–2) + ... + L[i+1].
# On vérifie ci-après que cette dernière égalité est un invariant de boucle.
        p ← p*a + L[i]
        # A la fin de la boucle p = L[n]a^(n–i) + L[n-1]a^(n–i–1) + ... + L[i+1]a + L[i].
        # i change ensuite de valeur en décroissant de 1, donc au début de la boucle suivante,
        # avec la nouvelle valeur de i, on a à nouveau p = L[n]a^(n–i–1) + L[n-1]a^(n–i–2) + ... + L[i+1].
finfaire
# A la fin de la dernière boucle, i = 0 et p = L[n]a^(n–i) + L[n-1]a^(n–i–1) + ... + L[i+1]a + L[i].
# donc p = L[n]a^n + L[n-1]a^(n–1) + ... + L[1]a + L[0] = P(a)
```

De plus, les valeurs successives prises par p au cours du calcul ne sont autres que les coefficients du polynôme Q, quotient de P par $X - a$, la valeur finale de p étant P(a) = R reste de cette division. En effet, si on indice les valeurs de p par les valeurs de l'indice de boucle i, on a :

$p_n = \lambda_n$
$\forall\, i \in [\![\, 0, n-1 \,]\!],\ p_i = p_{i+1}a + \lambda_i$

Si on pose $Q = p_n X^{n-1} + ... + p_1$ et $R = p_0$, on constate que les relations vérifiées par les p_i sont précisément celles qui permettent d'écrire :

$$(X - a)Q + R = (X - a)(p_n X^{n-1} + ... + p_i X^{i-1} + ... + p_1) + p_0$$
$$= p_n X^n + (p_{n-1} - ap_n)X^{n-1} + ... + (p_i - ap_{i+1})X^i + ... + (p_1 - ap_2)X + p_0 - ap_1$$

$$= \sum_{k=0}^{n} \lambda_k X^k = P$$

4- Arrêt d'un algorithme

Un algorithme doit effectuer un nombre fini d'instructions avant de s'arrêter. Comment en être sûr ? La seule chose qui pourrait empêcher un algorithme de se terminer est une instruction itérative du type tant que <condition> faire où la condition ne prend jamais la valeur fausse. Il n'existe malheureusement pas de procédure universelle permettant de décider si une telle boucle s'arrête ou non. Il est évident, par exemple, que l'algorithme de la multiplication égyptienne se termine pour tout couple de données entières (A, B) car la variable A prend successivement des valeurs entières strictement décroissantes. Par contre, on ignore si l'algorithme de la suite de Collatz se termine pour toute valeur de a_0.

IV : Bases de données

Ce paragraphe a pour but de se familiariser avec le vocabulaire utilisé dans les systèmes de gestion de bases de données, dits bases de données relationnelles.

1- Attributs et schémas relationnels

Nous prendrons comme exemple de base de données la gestion d'une bibliothèque. Les données concernées sont :

- la liste des livres que possèdent cette bibliothèque,
- la liste des emprunteurs inscrits à cette bibliothèque,
- l'enregistrement des livres présents en rayon, ceux qui sont empruntés et ceux qui sont en réserve.

La liste des livres possédés par la bibliothèque doit contenir un certain nombre d'informations. Chaque livre est caractérisé par des **attributs** que sont : son *titre*, son *auteur*, son *éditeur*, son *année d'édition*, sa *cote*. La cote a pour but d'identifier chaque exemplaire de livre de façon unique alors qu'il peut y avoir plusieurs exemplaires du même livre. La cote sert donc de clef d'identification unique pour chaque exemplaire possédé par la bibliothèque. On la qualifie de **clef primaire**. On affecte un nom conventionnel à chacun de ces attributs (par exemple ici : titre, auteur, edition, annee_edition, cote).

Chaque attribut est affecté à un type d'information présent dans la base de données. Les valeurs que peut prendre cette information décrit un **domaine**, propre à chaque attribut, et noté dom(A) si A est le nom de l'attribut considéré. Ainsi dom(titre) = dom(auteur) = dom(editeur) = {chaînes de caractères}. dom(annee_edition) = $\mathbb{N}$. dom(cote) est une chaîne de caractères conventionnels, propres à distinguer chaque cote (par exemple 3 lettres suivi de 4 chiffres).

Passons maintenant à la liste des emprunteurs. Un emprunteur est défini par les attributs que sont : *nom*, *prénom*, *adresse*, *numéro d'inscription*, attributs que nous nommerons nom, prenom, adresse, num_inscription.

Enfin, la situation d'un livre est définie par les attributs que sont : sa *cote*, son *état* (en rayon, emprunté, en réserve), la *date d'emprunt* et le *numéro d'inscription* de son emprunteur s'il est

emprunté. On peut convenir que ces deux dernières données sont nulles si le livre n'est pas emprunté. Les attributs seront nommés ici cote, etat, date_emprunt, num_inscription.

On dispose donc d'un ensemble complet d'attributs :

att = {titre, auteur, editeur, annee_edition, cote, nom, prenom, adresse, date_emprunt, num_inscription, etat}.

Une partie U de cet ensemble d'attributs définit un **schéma relationnel**. Nous avons précédemment implicitement défini de la sorte les schémas relationnels suivants, chacun correspondant à une catégorie de données qu'on se propose de traiter :

Livre	par la partie U1 = {titre, auteur, editeur, annee_edition, cote}.
Emprunteur	par la partie U2 = {nom, prenom, adresse, num_inscription}.
Situation	par la partie U3 = {cote, etat, date_emprunt, num_inscription}.

On peut préciser le domaine de chaque attribut :

Livre(titre : chaîne de caractères, auteur : chaîne de caractères, editeur : chaîne de caractères, annee_edition : entier, cote : chaîne de caractères)

On remarque que cote est un attribut commun à Livre et à Situation, et que num_inscription est commun à Emprunteur et à Situation. Si on veut lier plus spécifiquement l'attribut au schéma relationnel dont il est issu, on précisera Livre.cote ou Situation.cote, et Emprunteur.num_inscription ou Situation.num_emprunteur. Il est également possible de donner des noms différents d'attributs, d'une part à la cote du schéma Livre, d'autre part à la cote du schéma Situation. Il est ensuite possible de faire se correspondre ces deux noms différents (voir plus loin la notion de jointure).

Lors de la création d'une base de données, la première chose qui est demandée à l'utilisateur est de créer la liste des attributs et les schémas relationnels qu'il souhaite utiliser. L'ensemble des schémas relationnels ainsi choisis s'appelle un **schéma de base de données**.

BIBLIOTHEQUE =
{Livre[titre, auteur, editeur, date_edition, cote],
Emprunteur[nom, prenom, adresse, num_inscription]
Situation[cote, etat, date_emprunt, num_inscription]}

Ce schéma décrit la structure qui sera utilisée dans la base de donnée.

Puis vient la phase d'enregistrement des données elles-mêmes.

2- Données et relations

Un livre sera représenté dans la base de données par l'enregistrement d'un quintuplet de données relatives aux attributs U1, par exemple :

<Les misérables, Victor Hugo, Editions Dubois, 2002, HUG-0145>

si l'ordre des attributs a été précisé, ou sinon sous la forme :

<titre : Les misérables, auteur : Victor Hugo, éditeur : Editions Dubois, annee_edition : 2002, cote : HUG-0145>

Un emprunteur sera représenté dans la base de données par un quadruplet de données relatives aux attributs U2, par exemple :

<nom : Bonnot, prénom : Jean, adresse : 2 rue du Pont 21000 Dijon, num_inscription : 3017>

La situation d'un livre est donnée par un quadruplet de données relatives aux attributs U3, par exemple :

<cote: HUG-0145, état : emprunté, date_emprunt : 10/05/2020, num_inscription : 3017>

La bibliothèque dispose évidemment de plusieurs livres. Pour cela, on dresse la liste des enregistrements décrivant chaque livre. Cette liste s'appelle une **relation** ou une **instance** sur le schéma relationnel Livre. Elle forme une partie de l'ensemble dom(titre) × dom(auteur) × dom (editeur) × dom(annee_edition) × dom(cote). Une relation peut être visualisée comme une table à deux entrées. On affecte à chaque attribut du schéma relationnel une colonne, et on fait figurer en *i*-ème ligne un *n*-uplet représentant le *i*-ème enregistrement de la relation. A l'intersection de la *i*-ème ligne et de la *j*-ème colonne figure donc la valeur du *j*-ème attribut pour le *i*-ème élément. Par exemple :

Livre

titre	**auteur**	**éditeur**	**annee_edition**	**cote**
<Les misérables,	Victor Hugo,	Editions Dubois,	2002,	HUG-0145>
<Les misérables,	Victor Hugo,	Editions Dubois,	2002,	HUG-0146>
<Boule de Suif,	Guy de Maupassant,	Editions Durand,	2005,	MAU-0238>
<Quatre-Vingt-Treize,	Victor Hugo,	Editions Dubois,	2002,	HUG-0202>
<Germinal,	Emile Zola,	Editions Martin,	2004,	ZOL-0134>
etc...				

Les *n*-uplets figurant dans une relation doivent posséder une clef qui permet de les identifier de manière unique. On exclut en effet d'avoir deux enregistrements rigoureusement identiques (il y aurait redondance). C'est le logiciel de gestion de base de données qui, au moment où l'on procède à l'enregistrement d'un nouveau *n*-uplet, doit s'assurer de cette non-redondance. Dans l'exemple précédent, c'est la cote qui joue le rôle de clef. Dans le cas du schéma Emprunteur, la clef est jouée par le numéro d'inscription de l'utilisateur, mais on peut aussi choisir comme clef le triplet <nom, prenom, adresse> attendu qu'a priori, une seule personne portant un nom et un prénom donné habite à l'adresse indiquée. Dans le cas du schéma Situation, la clef est jouée par la cote du livre considéré. Ci-dessus, il y a deux exemplaires du livre "Les misérables", affectés de cotes différentes. Les clefs sont choisies si possible de façon à posséder un nombre minimal d'attributs permettant d'identifier l'enregistrement de manière unique.

Une **base de données** ou **instance d'un schéma de base de données** permet alors d'associer à chaque schéma relationnel une instance sur celui-ci. C'est une réalisation concrète du schéma de base de données précédemment défini.

Ayant défini ces schémas relationnels, puis enregistré les relations correspondantes, l'utilisateur va vouloir interroger sa base de données en lui adressant des requêtes. Ces requêtes sont en fait des opérations algébriques sur la base de données.

3- Opérations sur la base de données

Il existe en premier lieu des opérations purement ensemblistes sur des relations ayant même schéma relationnel. Il s'agit de :

La réunion :

On peut réunir deux relations R et S en une seule R ∪ S. Les enregistrements de R ∪ S sont les *n*-uplets qui appartiennent à R ou à S. Cela se produit lorsqu'on veut fusionner en un seul fichier tous

les enregistrements provenants de fichiers différents. Un logiciel gérant des bases de données et qui permet la réunion de deux relations doit faire en sorte de supprimer les doublons provenant d'enregistrements qui figurent à la fois dans la relation R et dans la relation S, pour n'en garder qu'un seul.

L'intersection :
On peut intersecter deux relations R et S en une seule $R \cap S$. Les enregistrements de $R \cap S$ sont les *n*-uplets qui appartiennent à la fois à R et à S. On ne souhaite garder ici que les enregistrements communs à plusieurs fichiers. La relation résultante peut être vide.

La différence :
On peut effectuer la différence de deux relations R et S. La relation R – S ou R \ S est constituée des enregistrements qui appartiennent à R, mais pas à S.

Il existe aussi des opérateurs spécifiques aux bases de données. Les opérateurs les plus courants sont :
La sélection (ou restriction) :
Elle permet de sélectionner des lignes (i.e. des *n*-uplets) de la relation selon un critère donné, et d'éliminer les autres. La sélection s'applique sur une seule relation. On la représentera par le symbole σ, en précisant en indice les critères retenus. Ainsi, la sélection des lignes de la relation Livre ayant pour attribut auteur = "Victor Hugo" donne comme résultat :

<Les misérables, Victor Hugo, Editions Dubois, 2002, HUG-0145>
<Les misérables, Victor Hugo, Editions Dubois, 2002, HUG-0146>
<Quatre-Vingt-Treize, Victor Hugo, Editions Dubois, 2002, HUG-0202>
etc...

et le résultat obtenu se note :

$$\sigma_{\{auteur = "Victor\ Hugo"\}}(Livre)$$

On obtient une partie de la relation Livre qui correspond à l'ensemble suivant :

$$A = \{<t, "Victor\ Hugo", e, a, c> \in Livre\}$$

La condition de sélection peut être plus complexe, en portant sur plusieurs attributs.

La projection :
Elle permet de sélectionner des attributs de la relation (i.e des colonnes) et d'éliminer les autres. On la représentera par le symbole π, en indiquant en indice les attributs retenus. Dans l'exemple précédent, si on ne souhaite conserver que l'information concernant le titre et la cote des ouvrages obtenus par la sélection A, on appliquera une projection sur l'attribut titre et l'attribut cote :

$$\pi_{\{titre,cote\}}(A)$$

ce qui donne comme résultat :

<Les misérables, HUG-0145>
<Les misérables, HUG-0146>
<Quatre-Vingt-Treize, HUG-0202>
etc...

et correspond à l'ensemble :

$$\{<t, c> \mid \exists <e, a>, Livre(t,"Victor\ Hugo",e,a,c)\}$$

Ci-dessus, la notation Livre(t,"Victor Hugo",e,a,c) désigne le prédicat $(t, "Victor\ Hugo", e, a, c) \in Livre$ qui prend la valeur Vrai ou Faux selon que le *n*-uplet <t, "Victor Hugo", e, a, c> appartient ou non à Livre. On peut appliquer à ces prédicats les opérateurs logiques usuels ("et", "ou", "non", voire même

"implique" et "équivaut à"), ainsi que les quantificateurs "il existe" et "quel que soit". On effectue alors du **calcul relationnel**.

La jointure :
Elle permet de combiner les informations de plusieurs relations, ayant des noms d'attributs communs. Elle se note $\bowtie$. Considérons par exemple les schémas de relation Livre et Situation. Si l'on veut savoir si tel livre, identifié par son titre et son auteur, est disponible, il faudra rechercher les cotes de ces livres dans la relation Livre, puis regarder dans la relation Situation si parmi les cotes retenues, l'une d'entre elles possède un attribut etat dont la valeur soit "en rayon". Pour cela, il faut relier Livre à Situation au moyen de l'attribut commun cote. C'est là le rôle de l'opérateur de jointure.

Si Rel1 et Rel2 sont deux relations ayant pour ensemble d'attributs respectivement Att1 et Att2, alors Rel1 $\bowtie$ Rel2 est la relation dont l'ensemble d'attributs est Att1 $\cup$ Att2 et dont les éléments sont les *n*-uplets *t* tels que $t_{|Att1}$ soit élément de Rel1, et $t_{|Att2}$ soit élément de Rel2. Les valeurs des attributs de *t* communs à Att1 et Att2 sont nécessairement les mêmes.

Dans l'exemple envisagé, la jointure de Livre par Situation donnera une unique relation recensant tous les livres ainsi que leur situation. On pourra alors sélectionner ceux qui sont en rayon et qui répondent au titre et à l'auteur voulu.

Si Att1 = Att2, alors Rel1 $\bowtie$ Rel2 n'est autre que l'intersection Rel1 $\cap$ Rel2.
Si Att1 et Att2 sont disjoints, alors Rel1 $\bowtie$ Rel2 n'est autre que le produit cartésien Rel1 $\times$ Rel2, dont les éléments sont constitués de *n*-uplets dont la première partie appartient à Rel1, et la deuxième à Rel2.

On peut également effectuer une jointure au moyen d'attributs portant des noms différents, à condition qu'ils aient le même domaine. Il suffit de préciser en option quels sont les attributs à identifier.

On peut vérifier que la jointure est associative. La jointure de Rel1 avec la relation {< >} constituée d'une unique ligne sans attribut redonne Rel1.

Le renommage :
Cette opérateur permet de renommer un attribut. Pour effectuer une jointure entre deux relations, il est nécessaire en effet que celles-ci aient un attribut en commun. Si ce n'est pas le cas, il est possible de faire correspondre un attribut de la première relation avec un attribut de la seconde en renommant l'un des attributs. Notons ρ le renommage. Alors $\rho_{\{attribut1 \rightarrow attribut2\}}$(Nomrelation) désigne l'opérateur qui renomme dans la relation Nomrelation l'attribut1 en attribut2.

Plus généralement, si Nomrelation est une relation ayant les attributs A1, A2, A3, ..., alors

$$\rho_{\{A1 \rightarrow B1,\ A2 \rightarrow B2\}}(\text{Nomrelation})$$

crée une relation ayant les attributs B1, B2, A3, ... et possédant les éléments *t* pour lesquels il existe *u* dans NomRelation tels que $t(Bi) = u(Ai)$, pour $i = 1, 2$, et $t(Ai) = u(Ai)$ pour $i \geq 3$.

La division cartésienne :
Soient S et R deux relations, telles que le schéma d'attributs Att2 de S soit inclus dans le schéma d'attributs Att1 de R. La division de la relation R par la relation S est une relation T = R/S dont le

schéma d'attributs est la différence Att1 \ Att2. Ses enregistrements *t* sont tels que la concaténation de *t* avec n'importe quel enregistrement de S donne un enregistrement de R. Par exemple, si R a pour attributs (auteur, editeur) et S a pour attribut (editeur), alors R/S donne la liste des auteurs publiés par tous les éditeurs.

4- Les requêtes et le langage SQL

Les opérateurs que nous avons vus sous forme algébrique dans le paragraphe précédent permettent de décrire l'ensemble des réponses à une requête sur la base des données. La combinaison de ces opérateurs forment l'**algèbre relationnelle**. Une requête consiste en la formulation d'une recherche de données à partir de critères précis. La réponse à cette requête est un ensemble, fourni par le logiciel de traitement de données en réponse à la requête demandée. On appelle cette réponse une **vue** de la base de données. Il s'agit d'une relation, mais issue d'un calcul et non physiquement présent sur une disque ou une mémoire. Cela signifie que, si les relations initiales sont modifiées par ajout, suppression ou modification d'enregistrements, les vues devront être recalculées pour être à jour. Néanmoins, une vue peut être réutilisée à son tour comme n'importe quelle autre relation pour définir d'autres vues ou répondre à d'autres requêtes.

Les requêtes dans un logiciel donné sont décrites dans un langage compréhensible par ce logiciel. C'est le cas du langage SQL dont nous donnerons quelques éléments. La plupart du temps, l'utilisateur n'a pas à connaître ce langage car une interface sert d'intermédiaire entre l'utilisateur et la base de donnée, par exemple par l'intermédiaire d'un formulaire à remplir. Ce formulaire est traduit par l'interface en instructions SQL à exécuter par le serveur de données pour répondre à la requête. Néanmoins, les logiciels de traitement des données offrent généralement la possibilité d'adresser des requêtes directement en langage SQL, ce qui permet de formuler exactement les critères souhaités, apportant des possibilités plus larges que la seule utilisation d'un formulaire dont le cadre d'utilisation est parfois limité. Signalons également l'architecture dite trois-tiers, séparée entre une interface accessible à l'utilisateur qui saisit ses requêtes sous forme conviviale et affiche les vues demandées, un logiciel intermédiaire qui effectue la partie logique du traitement (traduction des demandes en SQL et traitement des données recherchées sur le serveur) et le serveur de données lui-même. Si le serveur est par exemple unique, les logiciels intermédiaires peuvent être localisés en plusieurs pays du monde, ce qui permet une souplesse de fonctionnement et décharge le serveur d'une partie du traitement. De plus, on peut modifier le module de traduction des requêtes ou de présentation des réponses pour l'améliorer sans que le serveur gérant les données elles-mêmes n'en soit affecté, pas plus que ne l'est l'outil utilisé par le client.

Nous nous bornerons à donner la syntaxe de quelques commandes SQL relatives à la consultation d'une base de données. Il existe évidemment des commandes pour créer une base de données, y insérer des éléments ou les supprimer, ou modifier ces éléments, ainsi que des commandes pour gérer les droits d'accès aux bases de données, mais celles-ci sont en général également disponibles pour l'utilisateur au moyen d'une interface dont les menus permettent de gérer ces commandes.

La structure générale la plus élémentaire d'une commande SQL relative à une requête a la forme suivante :

SELECT *liste des attributs*	// on opère ici une projection π sur les attributs voulus
FROM *liste des relations*	// on indique ici les relations utilisées
WHERE *liste des conditions*	// on opère ici une sélection σ selon les critères choisis

Voyons comment se traduisent en SQL les opérateurs algébrique rencontrés auparavant :

Union : $R \cup S$

```
SELECT * FROM R
UNION
SELECT * FROM S
```

Intersection : $R \cap S$

```
SELECT * FROM R
INTERSECT
SELECT * FROM S
```

Produit cartésien : $R \times S$

```
SELECT *
FROM R,S
```

Différence : $R - S$

```
SELECT *
FROM  R
EXCEPT SELECT * FROM S
```

Projection : $\pi_{\{nomatt1,nomatt2\}}(R)$

```
SELECT nomatt1, nomatt2
FROM R
```

Sélection : $\sigma_{\{att1="blabla", att2="blibli\}}(R)$

```
SELECT *
FROM R
WHERE att1="blabla" AND att2="blibli"
```

Jointure : $R \bowtie S$ où la jointure se fait selon l'attribut commun att

```
SELECT  *
FROM  R JOINT S ON R.att = S.att
```

ou encore :

```
SELECT  *
FROM  R, S
WHERE R.att = S.att
```

ou enfin :

```
SELECT  *
FROM  R, S
WHERE R.att1 = S.att2
```

si les noms des attributs à faire correspondre sont respectivement att1 dans la relation R, et att2 dans la relation S.

Le langage possède également des commandes telles que IN permettant de tester l'appartenance d'un élément à un ensemble, ou BETWEEN permettant de sélectionner les éléments numériques appartenant à un intervalle. Il possède aussi des fonctions dites d'agrégation, qui calcule le

minimum, le maximum, la somme, la moyenne ou le nombre d'éléments d'une suite (MIN, MAX, SUM, AVG, COUNT). Par exemple, le nombre d'éléments dans la relation R est donnée par :

```
SELECT  COUNT(*)
FROM  R
```

Il peut exister plusieurs formules de l'algèbre relationnelle et donc plusieurs commandes SQL répondant à une même requête du calcul propositionnel. Chacune de ces formules correspond à un algorithme qui sera plus ou moins efficace en temps de calcul ou en espace mémoire occupé.

4- Exemples

a) Quels sont les livres de Victor Hugo que possède la bibliothèque ?

Cette requête ne nécessite que la consultation de la relation Livre. Elle consiste simplement à sélectionner dans Livre les enregistrements dont l'attribut auteur prend la valeur "Victor Hugo". On cherche donc :

$$\sigma_{\{auteur = "Victor\ Hugo"\}}(Livre)$$

et la commande SQL correspondante est :

```
SELECT *
FROM Livre
WHERE auteur = "Victor Hugo"
```

Le résultat de cette requête est l'ensemble :

{<t, "Victor Hugo", e, d, c> | Livre(t, "Victor Hugo", e, d, c)}

Cet ensemble est ici :

```
<Les misérables,      Victor Hugo,    Editions Dubois,    2002,  HUG-0145>
<Les misérables,      Victor Hugo,    Editions Dubois,    2002,  HUG-0146>
<Quatre-Vingt-Treize, Victor Hugo,    Editions Dubois,    2002,  HUG-0202>
etc...
```

On peut aussi effectuer une recherche sur une partie du nom de l'auteur, en utilisant le symbole % qui remplace toute suite de lettres. La commande suivante :

```
SELECT *
FROM Livre
WHERE auteur LIKE "%Hug%"
```

cherchera tous les livres dont l'auteur possède la suite de lettres HUG, quel que soit l'endroit où se situe cette suite de lettres dans le nom d'auteur.

On peut aussi demander à ce que le résultat de la requête soit affiché par ordre croissant d'un des champs, par exemple le titre :

```
SELECT *
FROM Livre
WHERE auteur LIKE "%Hug%"
ORDER BY titre
```

b) Quels sont les titres de livres de Victor Hugo que possède la bibliothèque ?

C'est une variante de la requête précédente, où l'on ne s'intéresse qu'aux titres. Il suffit d'opérer une projection selon l'attribut titre sur la vue trouvée en **a)** :

$$\pi_{\{titre\}}(\sigma_{\{auteur = "Victor\ Hugo"\}}(Livre))$$

et la commande SQL correspondante est :

SELECT titre
FROM Livre
WHERE auteur = "Victor Hugo"

La vue des réponses est :

{<t> | ∃ e, a, c, Livre(t, "Victor Hugo", e, a, c)}

Dans l'expression ∃ e, a, c, Livre(t, "Victor Hugo", e, a, c), la variable t non affectée d'un quantificateur, s'appelle **variable libre**.

c) Quels sont les livres de Victor Hugo qui sont empruntés ?

Cette requête nécessite la consultation des relations Livre et Situation, et la satisfaction du prédicat :

Livre(t, "Victor Hugo", e, a, c) et Situation(c, "emprunté", d, n)

On notera la variable commune c permettant que ce soit le même livre qui soit considéré dans les deux instances. On doit donc effectuer la jointure de Livre et de Situation, puis sélectionner les enregistrements dont auteur vaut "Victor Hugo" et dont etat vaut "emprunté". Si on souhaite connaître uniquement la cote et le numéro d'inscription de l'emprunteur, on conclura par une projection sur ces deux attributs. On obtiendra alors la vue :

{<c, n> | ∃ t, e, a, d, Livre(t, "Victor Hugo", e, a, c) et Situation(c, "emprunté", d, n)}

En algèbre relationnelle, cette vue est désignée par :

$\pi_{\{cote,num\_inscription\}}(\sigma_{\{auteur = "Victor Hugo", etat = "emprunté"\}}(Livre \bowtie Situation))$

et s'obtiendra en SQL au moyen de la commande :

SELECT Situation.cote, num_inscription
FROM Livre JOIN Situation ON Livre.cote = Situation.Cote
WHERE auteur = "Victor Hugo" AND etat= "emprunté"

d) Quels exemplaires du livre "Les misérables" sont en rayon ?

La démarche est analogue au **c)** sauf qu'ici, on s'intéresse à la cote des ouvrages disponibles. La liste des réponses peut être vide.

SELECT Livre.cote
FROM Livre JOIN Situation ON Livre.cote = Situation.Cote
WHERE titre = "Les misérables" AND etat= "en rayon"

e) Quelles sont les adresses des emprunteurs d'un livre de Victor Hugo ?

On doit pour cela consulter les trois instances. On effectue une jointure entre Livre et Situation selon l'attribut cote, puis une jointure avec Emprunteur selon l'attribut num_inscription. On sélectionne ensuite les enregistrements dont la valeur de auteur est "Victor Hugo" et celle de etat est "emprunté", puis on projette selon l'attribut adresse :

$\pi_{\{adresse\}}(\sigma_{\{auteur = "Victor Hugo", etat = "emprunté"\}}(Livre \bowtie Situation \bowtie Emprunteur))$

On obtient la vue :

{<adr> | ∃ t, e, a, c, d, n, nm, pr, n Livre(t, "Victor Hugo", e, a, c) et Situation(c, "emprunté", d, n) et Emprunteur(nm ,pr, adr, n)}

La commande SQL est :

SELECT adresse
FROM Livre JOIN Situation ON Livre.cote = Situation.Cote JOIN Emprunteur ON Situation.num_inscription = Emprunteur.num_inscription
WHERE auteur = "Victor Hugo" AND etat= "emprunté"

La façon dont cette requête est traitée peut modifier notablement le temps de calcul. Si on effectue la jointure des trois relations, on risque d'obtenir une relation dont la taille est conséquente. Dans le cas présent, il vaut mieux d'abord effectuer la jointure entre Livre et Situation, puis sélectionner dans cette jointure les livres dont l'auteur est Victor Hugo, puis seulement après, effectuer la jointure entre la relation résultat obtenue et la relation Emprunteur avant d'effectuer une sélection sur l'état.

$$\pi_{\{adresse\}}(\sigma_{\{etat = "emprunté"\}}(\sigma_{\{auteur = "Victor Hugo"\}}(Livre \bowtie Situation)) \bowtie Emprunteur))$$

En SQL, on va donc créer une vue obtenue par jointure de Livre et Situation, à laquelle on applique la sélection sur l'auteur :

```
CREATE VIEW Vue AS
SELECT * FROM Livre JOIN Situation ON Livre.cote = Situation.Cote
WHERE  auteur = "Victor Hugo";
```

Puis, on effectue la jointure entre le résultat précédent et la relation Emprunteur :

```
SELECT adresse
FROM Vue
JOIN Emprunteur ON vue.num_inscription = Emprunteur.num_inscription
WHERE etat= "emprunté";
```

La démarche à privilégier vise à optimiser le temps de calcul, et s'apparente à celle du calcul de complexité d'un algorithme.

f) Quels sont les livres empruntés par l'emprunteur dénommé Jean Bonnot ? :

On procède comme au **e)** :

```
SELECT Livre.cote, titre, auteur
FROM Livre JOIN Situation ON Livre.cote = Situation.Cote JOIN Emprunteur ON Situation.num_inscription = Emprunteur.num_inscription
WHERE nom = "Bonnot" AND prenom= "Jean" AND etat = "emprunté"
```

g) Quels sont les livres qui ne sont pas empruntés ?

Ils peuvent être en rayon, ou en réserve. On cherche :

rel = {<c> | ∃ d, n, Situation(c, "en rayon", d, n) ou Situation(c, "en réserve", d, n)}

La commande SQL est :

```
SELECT titre,auteur,etat
FROM Livre JOIN Situation ON Livre.cote=Situation.cote
WHERE etat="en rayon" OR etat="en réserve";
```

h) Quelles sont les cotes des livres dont l'édition est comprise entre 2000 et 2010 ?

```
SELECT cote
FROM Livre
WHERE annee_edition BETWEEN 2000 AND 2010
```

Exercices

1- Enoncés

Exo.1) Ecrire des fonctions **inttobin** et **bintoint**, convertissant respectivement un nombre entier en sa représentation binaire, et une représentation binaire en l'entier correspondant. Les représentations binaires, constituées d'une suite de 0 ou de 1, seront simplement fournies par les nombres décimaux

formés des mêmes chiffres. Par exemple, la fonction **inttobin** appliquée à l'entier 6 donnera comme résultat le nombre entier 110. Les procédures auront donc toutes comme paramètre un entier (entier constitué uniquement de chiffres 0 ou 1 pour la fonction **bintoint**) et leur résultat sera un entier (entier constitué uniquement de chiffres 0 ou 1 pour la fonction **inttobin**). Aucun tableau ne doit être utilisé dans ces procédures.

Exo.2) Ecrire un algorithme qui, étant donné un nombre entier *n*, calcule si le nombre *n* possède deux 0 consécutifs ou plus dans son développement binaire. Tel est le cas de $354 = 101100010_b$, mais pas de $181 = 10110101_b$.

Exo.3) Dans plusieurs langages de programmation, il existe une instruction break qui met fin à la boucle itérative qui la contient. Dans l'exemple qui suit, B et C sont des expressions booléennes, et R, S, T des suites d'instructions. Si la condition C est vraie après exécution des instructions R, on exécute les instructions S, mais on termine la boucle sans exécuter les instructions T. On demande d'écrire un programme équivalent au suivant, mais sans utiliser l'instruction break :

```
tant que B faire
        R
        si C alors S ; break finsi
        T
finfaire
```

Exo.4) Que fait l'algorithme suivant, où A est un tableau d'entiers dont les éléments sont indicés de 1 à *n*, et *x* un entier donné ?

```
i ← 1
j ← n
tant que i < j faire
        si A[i] > x alors temp ← A[j]
                          A[j] ← A[i]
                          A[i] ← temp
                          j ← j-1
                  sinon i ← i+1
        finsi
finfaire
```

Exo.5) Dans la programmation d'un jeu de Mastermind, on est amené à programmer le problème suivant. On se donne un tableau C de quatre chiffres choisis parmi {1, 2, 3, 4, 5, 6}, par exemple [6, 3, 2, 2]. On se donne également un autre tableau R de quatre chiffres dans le même ensemble, par exemple [6, 2, 1, 6]. On suppose que les indices des tableaux varient de 1 à 4.

a) Le nombre de bien placés est le nombre d'indices *i* tels que C[*i*] = R[*i*]. Dans l'exemple du préambule, il n'y a qu'un seul bien placé, le premier 6. Etant donnés C et R, écrire une suite d'instructions donnant le nombre BP de bien placés.

b) Le nombre de mal placés est le nombre d'indices *i* tels que R[*i*] soit égal à un C[*j*], un même C[*j*] ne pouvant être utilisé que pour un seul R[*i*], les biens placés étant exclus. Dans l'exemple du préambule, il y a un seul mal placé, le 2. Le dernier 6 n'est pas considéré comme mal placé puisque le 6 du tableau C correspond au 6 bien placé du tableau R. De même, il n'y a qu'un seul 2 mal placé. Etant donnés C et R, on demande une suite d'instructions donnant le nombre MP de mal placés. Dans le programme suivant, dès qu'un élément du tableau C est utilisé pour être associé à un élément mal placé du tableau R, il est mis à 0 pour ne pas être utilisé ultérieurement. (Si on ne souhaite pas modifier C, alors on copie C dans un autre tableau et c'est ce dernier qui est traité).

Cependant, le programme est incorrect. Expliquer pourquoi et le corriger pour qu'il fonctionne correctement :

```
MP ← 0:
pour i ← 1 à 4 faire
        j ← 1
        tant que (j<4) et (R[i]≠C[j]) faire j ← j+1 finfaire
        si (R[i] = C[j]) et (R[i]≠C[i]) alors
            MP ← MP + 1
            C[j] ← 0
        finsi
finfaire
```

Exo.6) Succession de combinaisons : Deux entiers $0 < p \leq n$ étant donnés, on souhaite afficher toutes les combinaisons de p éléments pris parmi n. Une telle combinaison C est représentée par un tableau de p entiers strictement croissants, compris entre 1 et n. On range les combinaisons par ordre lexicographique, i.e. l'ordre alphabétique, où les combinaisons sont considérées comme des mots dont les lettres sont les entiers de 1 à n. Par exemple, pour $p = 5$ et $n = 8$, les combinaisons sont, dans l'ordre :

[1, 2, 3, 4, 6]
[1, 2, 3, 4, 7]
[1, 2, 3, 4, 8]
[1, 2, 3, 5, 6]
[1, 2, 3, 5, 7]
[1, 2, 3, 5, 8]
...
[3, 4, 5, 6, 8]
[3, 4, 5, 7, 8]
[3, 4, 6, 7, 8]
[3, 5, 6, 7, 8]
[4, 5, 6, 7, 8]

On demande d'écrire une fonction Combsuiv(n,p,C) transformant la combinaison C en son successeur (s'il existe), et renvoyant vrai ou faux selon que ce successeur existe ou pas. En exécutant successivement la procédure Combsuiv tant qu'elle retourne le résultat vrai, et en affichant la valeur de C, on obtiendra l'affichage de toutes les combinaisons voulues.

Exo.7) Soit N un entier strictement positif, et L un tableau des N éléments de 1 à N dans le désordre (ou permutation des N éléments). On scanne L de gauche à droite autant de fois que nécessaire en effaçant les uns après les autres les éléments par ordre croissant de 1 à N, et ce, jusqu'à ce que tous les éléments de la liste aient été effacés. On marque un point chaque fois que l'on passe par un élément sans l'effacer. Le "désordre" de la liste est le score obtenu à la fin du procédé d'effacement[1]. Par exemple, le désordre de [3, 5, 2, 1, 4] est 8 puisque le processus passera par 3, 5, 2, 4 au premier passage en effaçant le 1, puis 3, 5, 4 au deuxième passage en effaçant le 2, puis 5 au troisième passage en effaçant le 3 et le 4. On demande d'écrire une procédure qui calcule le désordre d'une permutation. On conviendra que l'indice du tableau L est compris entre 1 et N.

[1] Cette notion, ainsi que l'exemple, a été introduite par E. Deutsch, Problem 10975, 111:7, Amer. Math. Monthly (2004), 628.

Exo.8) On suppose dans cet exercice que le logiciel de programmation est capable de manipuler de grands entiers (il en est de même dans l'exercice 9). On présente un test probabiliste permettant de déterminer si un nombre n est premier. Soit n un nombre impair. Posons $n - 1 = t \times 2^s$, avec t impair. Soit b compris entre 2 et $n - 1$. On dit que b et n réussissent le **test de Miller-Rabin** si $b^t \equiv 1 \bmod n$ ou si il existe r, $0 \leq r < s$ tel que $b^{t2^r} \equiv -1 \bmod n$. Dans le cas contraire, on dit que le test échoue[2].
On peut prouver que, si n est premier, le test réussit pour tout $b \in [\![2, n - 1]\!]$, mais si n n'est pas premier, le test échoue pour au moins 3/4 des nombres b. On choisit k nombres b au hasard. Si l'un des tests échoue, on est certain que n est composé. Si le test réussit, alors n est premier avec une probabilité d'erreur inférieure à $\frac{1}{4^k}$, soit $\frac{1}{10^{18}}$ si $k = 30$. La probabilité d'erreur est en fait certainement encore plus faible. Pour $b = 2$, 3, 5 ou 7 seulement, le seul nombre composé n inférieur à $2{,}5 \times 10^{10}$ qui passe le test est 3215031751. En outre, il a été prouvé que le test précédent serait déterministe (et non plus probabiliste) en un temps de calcul $O(\ln^2 n)$, à condition que la conjecture dite de Rieman soit vérifiée.

a) Ecrire une fonction decompose qui, en entrée, admet pour paramètre un entier p, et en sortie, fournit le couple constitué des deux éléments s et t tels que $p = t \times 2^s$, avec t impair.

b) Ecrire une fonction test, qui, en entrée, admet pour paramètre b et n, et donne comme résultat le booléen vrai ou faux suivant que le test a réussi ou pas. Du point de vue pratique, on calcule b^t modulo n. Si la valeur vaut 1 ou –1, le test est réussi. Sinon, on élève b^t au carré, puis encore au carré... jusqu'à obtenir –1 auquel cas le test est réussi pour ce b, ou bien jusqu'à ce qu'on ait obtenu un carré égal à 1, auquel cas le test a échoué, la valeur –1 ne pouvant plus être obtenue. Au pire, on s'arrêtera au bout de s boucles, le test échouant également dans ce cas.

c) Ecrire une fonction premier qui teste de façon probabiliste si un entier n est premier. Pour cela, on effectuera le test précédent avec au plus trente valeurs de b prises aléatoirement entre 2 et le minimum de n et 10^{10}. On supposera qu'on dispose d'une fonction min(x,y) donnant le minimum de x et y, et d'une fonction rand(x,y) donnant comme résultat un entier pseudo-aléatoire entre x et y.
On pourra appliquer ce test à $2^{2^5} + 1$ que Fermat croyait premier, affirmation qu'Euler a réfutée.

Exo.9) La recherche de diviseurs d'un entier composé n est une opération longue et difficile. La méthode naïve consistant à chercher les diviseurs potentiels de 2 jusqu'à $\sqrt{n}$ est beaucoup trop longue dès que n dépasse une vingtaine de chiffres. On décrit ici une méthode, dite **méthode rho de Pollard**. Elle consiste à choisir au hasard des nombres x_i, et à calculer les PGCD de $x_i - x_j$ et de n. Si un tel PGCD est différent de 1, on a trouvé un diviseur. Dans la pratique, les x_i ne sont pas aléatoires mais sont les termes d'une suite définie par récurrence : $x_{i+1} = f(x_i)$, où la fonction f n'est pas trop régulière. On prendra $f(x) = x^2 + 1 \bmod n$ qui se révèle un bon candidat. Pour éviter d'accumuler les valeurs x_i dans un tableau, on se contentera de calculer le PGCD de $x_{2i} - x_i$ et de n, et pour cela, on définira deux variables locales x et y, x prenant les valeurs x_i et y les valeurs x_{2i}. La méthode est assez efficace.
Programmer une fonction diviseur, de paramètre n supposé non premier, qui itère la méthode décrite précédemment jusqu'à ce qu'un diviseur soit trouvé. On supposera qu'il existe dans le langage considéré une fonction pgcd(a,b) calculant le PGCD des deux entiers a et b (voir L1/ARITHMTQ.PDF pour une description de l'algorithme).
On pourra appliquer cette fonction pour trouver un diviseur de $2^{2^5} + 1$.

[2] Neal Koblitz, A course in number theory and cryptography, Springer-Verlag, (1994), 130-134

Exo.10) Un cinéphile veut se créer une base de données lui permettant de stocker des informations relatives à tous les films qu'il voit. Proposer une liste d'attributs pertinents pour cette base. Donner la syntaxe d'une instruction SQL donnant la liste des films contenus dans la base et dont le réalisateur est Spielberg.

2- Solutions

Sol.1) Pour la procédure **inttobin**, on divise le paramètre n, de valeur initiale n0, successivement par 2, permettant de compléter le code binaire de droite à gauche. Le nombre p indique (sous forme d'un entier de la forme 100...0, autrement dit d'une puissance de 10) le chiffre binaire que l'on est en train de compléter. Le résultat est accumulé dans la variable b. Les variables n, b, p sont des variables locales propres à la fonction.

```
inttobin:=proc(n0)
n ← n0; b ← 0; p ← 1
tant que n ≠ 0 faire
        b ← b + (n mod 2)*p     # n mod 2 désigne le reste de la division de n par 2
        p ← 10*p
        n ← n//2                # n//2 désigne le quotient entier dans la division de n par 2
finfaire
b                               # on indique ici la variable qui servira de résultat à la fonction
finproc
```

On pourra vérifier que l'invariant de boucle est : $n0 = n \times 2^q + b$ où q est tel que $p = 10^q$, et b désigne l'entier dont le codage binaire est b, constitué d'au plus q chiffres. (Si b = 110, $b = 6$).

La procédure **bintoint** est identique, mais en intervertissant les rôles de 2 et 10 :

```
bintoint:=proc(b0)
b ← b0; n ← 0; p ← 1
tant que b ≠ 0 faire
        n ← n + (b mod 10)*p
        p ← 2*p
        b ← b//10
finfaire
n
finproc
```

Sol.2) On introduit une variable S qui calcule le nombre de 0 successifs rencontrés dans la décomposition binaire de n, en commençant par le chiffre des unités. On divise successivement n par 2. Si le résultat est pair, on incrémente S de 1. Si le résultat est impair, S est remis à 0. On s'arrête quand n est nul, ou quand S = 2.

```
S ← 0
tant que n ≠ 0 et que S ≤ 1 faire
        Si n%2=0 alors S ← S + 1        # n%2 désigne le reste dans la division de n par 2
                 sinon S ← 0
        finsi
        n ← n//2                        # n//2 est le quotient entier dans la division de n par 2
finfaire
```

Il suffit maintenant de regarder si S = 2.

On pourra vérifier que l'invariant de boucle est le suivant : le développement binaire de la valeur initiale de n est égal, à chaque itération, au développement binaire en cours de n, suivi de S chiffres 0, suivi d'une suite de chiffres initialement vide, ne commençant pas par 0 et ne contenant pas deux 0 de suite.

Sol.3) On peut utiliser une variable auxiliaire booléenne Termine, simulant l'instruction break :

```
Termine ← faux
tant que B et (non Termine) faire
        R
        si C alors S ; Termine ← vrai
            sinon T
        finsi
finfaire
```

L'instruction break peut faciliter l'écriture de programme. Par exemple, la recherche de l'indice d'un élément *k* dans un tableau T de *n* éléments, dont les éléments sont indicés de 0 à *n* – 1, peut s'écrire :

```
resultat ← – 1
pour i ← 1 à n – 1 faire
        si T[i] = k alors resultat ← i; break finsi
finfaire
```

L'indice cherché est dans la variable résultat (celle-ci valant – 1 si *k* n'est pas élément de T).

Sol.4) L'indice *i* croît à partir de 1, l'indice *j* décroît à partir de *n*, jusqu'à ce qu'ils atteignent une valeur commune.
Si on trouve un élément A[*i*] du tableau strictement supérieur à *x*, alors on permute les éléments A[*i*] et A[*j*], et on décrémente la variable *j*.
Si A[*i*] est inférieur ou égal à *x*, alors on augmente simplement *i* de 1.
L'algorithme a pour effet de placer à gauche du tableau les éléments inférieurs ou égaux à *x* et à droite les éléments strictement supérieurs à *x*. En effet, l'invariant de boucle est le suivant : les éléments A[1], ..., A[*i* – 1] sont inférieurs ou égaux à *x*. Les éléments A[*j* + 1], ..., A[*n*] sont strictement supérieurs à *x*.
Exemple : appliqué à [2, 5, 1, 6, 4, 3], on obtient d'abord {1, 2, 3} (pas nécessairement dans l'ordre), puis {4, 5, 6}.

Sol.5) On adopte la convention selon laquelle les indices des tableaux C et R varient de 1 à 4.
a)

```
BP←0
pour i ← 1 à 4 faire
        si C[i] = R[i] alors BP ← BP + 1 finsi
finfaire
```

b) Pour chaque élément R[*i*], on cherche un C[*j*] autre que C[*i*] qui puisse être associé à R[*i*]. Cependant, il convient aussi de vérifier que le C[*j*] ainsi trouvé n'est pas utilisé avec un bien placé R[*j*]. Si on prend C = [6, 3, 2, 2] et R = [6, 2, 1, 6], le programme associe à R[4] = 6 le C[1] = 6, et R[4] sera considéré comme mal placé de façon incorrecte. Pour corriger cette erreur, il suffit de vérifier que C[*j*] ≠ R[*j*].

```
MP ← 0:
pour i ← 1 à 4 faire
        j ← 1
        tant que (j<4) et (R[i]≠C[j] ou R[j]=C[j]) faire j ← j+1 finfaire
        si (R[i]=C[j]) et (R[i]≠C[i]) et (R[j]≠C[j]) alors
            MP ← MP + 1
            C[j] ← 0
        finsi
finfaire
```

L'invariant de la boucle *tant que* au cours de laquelle on traite R[*i*] est : aucun C[*k*], *k* < *j*, ne correspond au fait que R[*i*] est mal placé. Autrement dit : ∀ *k* < *j*, R[*i*] ≠ C[*k*] ou R[*k*] = C[*k*]. Par ailleurs, on peut économiser quelques calculs inutiles en testant avant la boucle *tant que* si R[*i*] est bien placé (i.e. R[*i*] = C[*i*]), auquel cas il n'a pas à être traité.

Sol.6) On convient que les indices des tableaux varient de 1 à *n*. Soit C une combinaison de *p* éléments. Partant de la droite, on regarde le nombre *h* (éventuellement nul) de valeurs du tableau C qui sont décrémentées de 1, en partant de *n*.
Si $h = p$, alors C est la dernière combinaison $[n - p + 1, ..., n - 1, n]$ et la fonction Combsuiv(n,p,C) renverra la valeur faux.
Sinon, C est de la forme $[..., m, n - h + 1, ..., n - 1, n]$, avec *m* en position $p - h$, mais avec $m < n - h$. Dans ce cas, la combinaison suivante est $[..., m + 1, m + 2, ..., m + h, m + h + 1]$. On cherche donc cet élément *m*, puis on modifie la suite du tableau.
Ci-dessous, les variables *h* et *m* ne sont pas explicitées. Dans la première boucle tant que, $h = p - i$ (initialement $h = 0$), et l'invariant de boucle est $C[k] = n - k + p$, pour *k* variant de $i + 1$ à *p*. On sort de la boucle avec cette condition vérifiée et l'alternative $C[i] \neq n + i - p$ ou $i = 1$. Par conséquent, si le test qui suit est vérifié, alors $C[k] = n - k + p$, pour *k* variant de *i* à *p* et $i = 1$, donc on a bien atteint la dernière combinaison, sinon, on est dans le cas où $m = C[i] < n + i - p = n - h$, et l'on passe à la combinaison suivante en augmentant *m* de 1 et en incrémentant cette valeur pour tous les éléments suivants du tableau.

```
Combsuiv:=proc(n,p,C)
i ← p
tant que (C[i]=n+i-p) et (i>1) faire i ← i-1 finfaire:
si C[i]=n+i-p alors faux
sinon
        C[i] ← C[i]+1:
        pour j ← i+1 à p faire C[j]←C[j-1]+1 finfaire
        vrai
finsi
finproc
```

Sol.7) Des est une variable décomptant le nombre de points marqués. Sa valeur finale sera le score demandé. *i* est un indice permettant de parcourir le tableau. *k* prend les valeurs de 1 à N : il décrit lq valeur des éléments que l'on efface les uns après les autres. On augmente le score Des de 1 à chaque fois que l'on passe sur un élément $L[i]$ supérieur à *k*. Quand on passe par la valeur *k*, cette valeur est incrémentée de 1. Il n'est pas utile dans l'algorithme d'effacer effectivement l'élément *k* du tableau, il suffit de ne pas tenir des éléments $L[i] < k$ lorsqu'on les rencontre puisqu'ils sont supposés avoir été effacés. Cela évite de modifier le tableau L. Si $i = N$, on atteint le bout du tableau, et on remet *i* à 1 afin de repartir au début du tableau.

```
i ← 1
Des ← 0
pour k ← 1 à N faire
        tant que L[i]≠k faire
                si L[i]>k alors Des ← Des+1 finsi
                i ← i+1
                si i=N+1 alors i ← 1 finsi
        finfaire
finfaire
```

Si on souhaite décrire un invariant de boucle, appelons *tour* le fait que *i* varie de 1 à N + 1 avant d'être remis à 1. Au cours d'un tour donné, introduisons la variable Dc comptant le nombre de points de désordre pris en compte uniquement lors de ce tour. Par définition, lorsqu'on parcourt le tableau pour chercher l'élément *k*, on a :

$i = Dc + \text{Card}\{j \mid j < i, L[j] < k\} + 1$

puisque les indices j vérifiant $j < i$, L[j] < k correspondent aux éléments supposés avoir été effacés, et les autres correspondent aux points décomptés dans Dc. Le + 1 final provient du fait qu'au début du tour, i vaut 1, alors que Dc est nul est que l'ensemble $\{j \mid j < i, L[j] < k\}$ est vide. On pourra vérifier que cette condition est précisément l'invariant de la boucle *tant que*, un autre invariant étant le fait que Des est la somme de Dc et des points de désordre obtenus lors des tours précédents (étudier tous les cas possibles : L[i] < k, L[i] > k et L[i] = k pour le premier invariant, ainsi que le passage de i par la valeur N + 1 pour le second). La valeur finale de Des sera donc bien le nombre de points de désordre cherché.

Sol.8) a) L'invariant de boucle dans l'algorithme ci-dessous est $n = 2^s t$, avec $s = 0$ et $t = n$ initialement. Si t est pair, on a $n = 2^{s+1} \times \frac{t}{2}$, d'où les nouvelles valeurs de s et t.

```
decompose := proc(n)
# s,t sont des variables locales à la fonction
s ← 0
t ← n
tant que t mod 2 = 0 faire s ← s+1;t ← t/2 finfaire
[s,t]
finproc
```

b) La boucle itérative utilisée ci-dessous peut être traitée plus simplement si le langage de programmation utilisé permet de combiner à la fois une instruction tant que avec une instruction pour, ou si le langage comporte une instruction break comme vu dans l'exercice 3). A défaut, on utilise un booléen termine indiquant quand la boucle est terminée. Par ailleurs, on supposera que b^t mod n est calculé conformément aux remarques concernant l'algorithme d'exponentiation rapide modulo n exposé plus haut dans ce cours.

```
test := proc(b, n)
# Les variables d, t, s, bb, i, termine sont locales à la fonction
d ← decompose(n-1)
s ← d[1]
t ← d[2]
bb ← b^t mod n
si bb = 1 ou bb = n-1 alors vrai
sinon
        termine ← faux
        r ← 0
        tant que non termine faire
                # on a l'invariant de boucle bb = b^(t 2^r)
                si bb ≠ 1 et bb ≠ n-1 et r < s-1 alors
                        bb ← bb^2 mod n
                        r ← r+1
                sinon # on a bb = 1 ou bb = n-1 ou bb = b^(t 2^(s-1))
                        termine ← vrai
                finsi
        finfaire
        si bb = n-1 alors vrai sinon faux finsi
finsi
finproc
```

c) Comme pour le b), selon le langage de programmation utilisé, la boucle ci-dessous peut être simplifiable. Par ailleurs, l'usage de variables locales permet que, même en cas d'homonymie (comme pour la variable termine par exemple), chaque variable est propre à la fonction dans laquelle elle est utilisée, et le logiciel de programmation est capable de les distinguer, sans confusion.

```
premier := proc(n)
```

```
# Les variables b, i, prem, termine sont locales à la fonction
termine ← faux
i ← 1
tant que non termine faire
        b ← rand(2, min(10000000000, n))
        prem ← test(b, n)
        i ← i+1
        si i=31 ou non prem alors termine ← vrai finsi
finfaire
prem
finproc
```

Sol.9)
```
diviseur := proc(n)
# variables locales PGCD,x,y
x ← 1
y ← 1
PGCD ← 1
tant que PGCD=1 ou PGCD=n faire
        x ← (x^2+1) mod n
        y ← (y^2+1) mod n
        y ← (y^2+1) mod n
        PGCD ← pgcd(x-y,n)
finfaire
PGCD
finproc:
```

L'algorithme donne 641 comme diviseur de $2^{2^5} + 1$. Avant d'appliquer la fonction sur un entier *n* quelconque, il convient d'avoir vérifié au préalable que *n* est composé (par exemple en lui appliquant l'algorithme de l'exercice précédent). Sur un nombre premier, en effet, l'algorithme tourne indéfiniment.

Sol.10) On peut proposer comme attributs utilisés dans la base de données, par exemple :

Titre	chaîne de caractères
Realisateur	chaîne de caractères
Annee	entier
Genre	chaîne de caractères
Acteurs	chaîne de caractères
Resume	chaîne de caractères

Si la base de données porte le nom film, la syntaxe demandée est :

```
SELECT *
FROM film
WHERE Realisateur LIKE "%Spielberg%"
```

◆