Chapitre 3

Notions d'algorithme et de programme

3.1 Introduction

Après avoir abordé l'historique de l'informatique et les outils électroniques de chaque génération, nous allons discuter dans ce chapitre les notions d'algorithmique.

Nous allons parler des éléments de base intervenant dans l'écriture d'un algorithme. Nous allons parler et classifier les données en types algorithmiques. Nous détaillerons ensuite chaque type, ses caractéristiques ainsi que les opérations possibles dans chaque type.

Nous verrons ensuite, les actions de base d'un algorithme que nous appellerons instructions. Nous discuterons de leurs classifications et nous survolerons quelques exemples d'actions simples.

A partir des éléments qu'on pourra voir dans ces chapitres, nous proposerons quelques problèmes simple à résoudre et nous dégagerons les algorithmes correspondants.

Nous clôturerons le chapitre par une conclusion et quelques exercices.

L'algorithmique est une discipline de l'informatique qui concerne les méthodes de résolution des problèmes à l'aide des machines informatiques. Cette discipline est plus ancienne des machines de l'informatique et de son évolution, car, de façon plus général, un algorithme représente une manière rigoureuse pour résoudre un problème donné. Une recette de cuisine peut être perçue comme un algorithme, qui, à partir d'un certain nombre d'ingrédient, essaye de préparer un plat donné en suivant des étapes bien déterminées. Les premiers algorithmes naissent en antiquité avec les *Babyloniens* au III^{e} siècle (AJC) qui décrivaient des méthodes de calcul et de résolution d'équations avec des exemples et puis avec les savants grecs notamment, *Euclide*, *Eratosthène*, *Archimède*, etc...

Le mot algorithme doit son nom au célèbre mathématicien perse *Abu Abdullah Muhammad ibn Musa al-Khwarizmi* datant du IX^e siècle.

Parmi les algorithmes connus bien avant l'informatique, on distingue :

L'algorithme d'*Euclide* qui calcul le pgcd [1] entre deux nombres entiers.

L'algorithme d'*Archimède* qui donne une approximation du nombre π.

L'algorithme d'*Eratosthène* pour trouver les nombres premiers.

L'algorithme d'*Averroes* qui utilise des procédés de calcul.

3.2 Notions préliminaires d'algorithme

3.2.1 Problème des sceaux

Supposant qu'on a le problème consistant à échanger les contenus de deux sceaux A et B. Comment procéder ?

1. On commencera par ajouter un troisième sceau C.
2. On videra le contenu de A dans C.
3. On versera le contenu de B dans A.
4. On versera le contenu de C dans B.

Pour ce petit problème trivial, la solution été d'énumérer les étapes nécessaires à la résolution de notre problème. Tout algorithme sera ensuite défini comme étant un ensemble fini d'opération qui se termine pour aboutir à une solution. Ces étapes doivent être en plus non ambigües.

On dira en plus pour toutes les étapes servant à résoudre un problème (même celles pour répondre à une énigme par exemple) qu'elles forment un algorithme de résolution de ce problème. On pourra parfois retrouver plusieurs algorithmes (ou suite d'étapes) pour résoudre le même problème, on optera le plus souvent pour la meilleure solution (moins d'étapes de construction, qui s'exécute rapidement, etc). L'énigme des missionnaires et des cannibales peuvent être perçue comme un problème à lequel il faut trouver une solution pour sauver les missionnaires. Résoudre une grille de Sudoku représente elle aussi un problème dont l'algorithme une fois établie, permettra de résoudre n'importe quelle grille par exemple.

1. plus grand diviseur commun

3.2.2 Définitions

Un algorithme est une suite d'instructions(actions) traduisant la solution d'un problème. Il sera traduit dans un langage donné et exécuté par un ordinateur, en un temps fini, pour arriver à un résultat donné, à partir d'une situation donnée. Il doit avoir les caractéristiques suivantes :

Lisible : L'algorithme doit être compréhensible même par un non informaticien.

De haut niveau : L'algorithme doit pouvoir être traduit dans n'importe quel langage de programmation.

Précis : Chaque élément de l'algorithme ne doit pas porter à confusion, il est donc important de lever toute ambiguïté.

Structuré : Un algorithme doit être composé de différentes parties facilement identifiables.

3.2.3 Structure d'un algorithme

Un algorithme est constitué de quatre parties :

Entrée des données : Informations utilisées pour résoudre un problème.

Sorties des résultats : Informations décrivant la solution d'un problème.

Lexique : Calcul intermédiaire.

Traitements : Instructions (ou actions) constituant le traitement.

Tout algorithme qu'on écrira dans ce cours aura le schéma type suivant :

Algorithme : Nom de l'algorithme
Déclaration des structures de données ;
Déclaration des fonctions ;
var Déclaration des variables ;
Début
    instruction 1;
    instruction 2;
    ...
    instruction n;
Fin.

Algorithme 1 : Schéma type d'un algorithme.

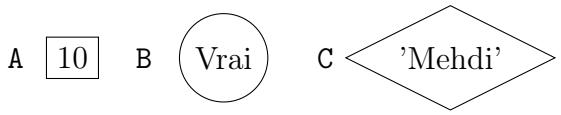

FIGURE 3.1 – A, B et C trois variables de type : numérique, logique et alphanumériques.

3.2.4 Étapes de conception d'un algorithme

Tout algorithme doit passer par les étapes suivantes avant d'être écrit :

Analyse / spécification : Dans cette étape on doit dégager avec précision les données en entrée, les résultats en sortie ainsi que la fonction principale du problème posé.

Conception : La conception consiste à déterminer les instructions (actions) permettant de traduire les données en entrée en résultats en sortie.

Écriture de l'algorithme : On devrait en fin traduire les étapes une et deux en langage algorithmique.

3.3 Éléments de base

En algorithmique une donnée est classée en trois types élémentaires :

1. Données numériques, eg : température, nombre de jours, etc...
2. Données alphanumériques, eg : un nom, une initiale d'un prénom, etc...
3. Données logiques, eg : état d'un interrupteur.

3.3.1 Données

Une donnée peut être considérée comme une boite ayant une **étiquette**, d'une certaine **forme (type)** et contenant une **valeur**. Donc selon les types qu'on a vu précédemment, cette forme va changer ainsi que les valeurs que va prendre cette variable.

Tel que montré dans la figure 3.1, les lettres A, B et C représentent les étiquettes des trois variables. Les valeurs sont à l'intérieur et les formes (on en a trois dans l'exemple) désignent les trois types : numériques, alphanumériques et logiques.

3.3.2 Types numériques

Une variable de type numérique est destinée à recevoir des nombres entiers ou des réels. Exemple :

Nombres entiers signés : -1, $+55$,$+1032$,...

Nombres entiers non signés : 0, 12, 26,...

Nombres réels : 1.2, 55.0, 1×10^{-3}, -2.25,...

On rencontre tous le temps des données de ce type comme les température d'un moteur (nombre réel), état d'un moteur (marche/arrêt), nombre de défaillance d'un moteur (nombre entier non singé) ou la caractéristique d'un matériau (solide, liquide, etc.).

3.3.2.1 Les entiers

Une variable de type entier est destinée à prendre des valeurs dans l'ensemble $\mathbb{Z}$. On pourra donc manipuler des nombres entiers non signés et non signés. Ces nombres seront représenté en binaire en complément à deux.

Sur ces nombres, on connait en plus des opérations traditionnelles (`+,-,*`), le quotient de la division naturelle (noté dans ce cours par `div`) et le reste de la division naturelle (notée `mod`), et l'opération d'exponentiation (notée `^`). Voici quelque exemples :

```
15 div 5 = 3
12 mod 2 = 0
17 mod 2 = 1
 2  ^  3 = 8
```

Quand on dispose de N bits, on pourra représenter 2^N nombre différents. Dans une représentation binaires non signées et si $N = 8$, on représentera $2^8 = 256$ nombres allant de $\{0, 1, \ldots, 255\}$. Par contre en représentation à complément à 2, on aura les nombres $\{-128, -127, \ldots, 0, 1, \ldots, 127\}$.

3.3.2.2 Les réels

Une variable de type réel prend ses valeurs dans l'ensemble $\mathbb{R}$. On connait deux façon de représenter ces nombres. La forme habituelle (2.6) et la forme scientifique (on dispose alors, d'un signe, d'une mantisse et d'un exposant : $\pm 3 \times 10^{-2}$).

Les opérations permises sur les réels sont : l'addition (`+`), la soustraction (`-`), la multiplication (`*`), la division (`/`) et la puissance (`^`).

3.3.3 Types alphanumériques

Ce type de données permet de représenter des caractères ou des chaines de caractères.

3.3.3.1 Les caractères

On notera ce type **car** dans ce cours d'algorithmique et on se servira pour représenter des variables qui prendront un seul caractère comme valeur. Voici quelques exemples de caractères :

— `'M', 'E', 'H'`, ...

— `'d', 'i', 'u'`, ...

— `'0','1','2','3','4'`, ...

— `'+','-','@', '*'`, ...

— `' '` : le caractère espace, ...

Notez dans ces caractères la présence de (`' '`), qui caractérisera les caractères. La table 3.1 montre quelques caractères de la table ASCII, avec leurs codes correspondants. L'ensemble des opérations sur les variables de type caractère sont : l'égalité (=), la différence notée ($\neq$ ou < >), supérieur (ou égal) (>, >=), inférieur (ou égal) (<, <=).

Les quatre dernières opérations s'opèrent à travers l'ordre des caractères dans la table ASCII (cf. table 3.1).

On sait par exemple que :

`'A'` $\neq$ `'l'`
`'a' > 'A' car 97 > 65`

3.3.3.2 Les chaînes de caractères

Une chaine de caractères est une suite de caractères. Elle peut être vide ou contenir un seul caractère. Exemple :

`"cours d'informatique"` **:** suite de caractères ;

`"C"` **:** chaîne composée d'un seul caractère ;

`""` **:** chaîne vide ;

Sur les variables de type chaîne de caractères, on connait, en plus des opérations permises sur les caractères, l'opération de concaténation notée (||). Cette opération permettras d'engendrer de nouvelles chaînes en mettant cote à cote deux chaînes. Exemple :

Code ASCII	Caractère	Code ASCII	Caractère
40	(	41	)
42	*	43	+
44	,	45	-
46	.	47	/
48	0	49	1
50	2	51	3
52	4	53	5
54	6	55	7
56	8	57	9
58	:	59	;
60	<	61	=
62	>	63	?
64	@	65	A
66	B	67	C
68	D	69	E
70	F	71	G
72	H	73	I
74	J	75	K
76	L	77	M
78	N	79	O
80	P	81	Q
82	R	83	S
84	T	85	U
86	V	87	W
98	b	99	c
100	d	101	e
102	f	103	g
104	h	105	i
106	j	107	k
108	l	109	m
110	n	111	o
112	p	113	q
114	r	115	s
116	t	117	u
118	v	119	w
120	x	121	y
123	z	124	|

TABLE 3.1 – Table ASCII pour quelques caractères.

A	B	Non A	A Et B	A Ou B
Faux	Faux	Vrai	Faux	Faux
Faux	Vrai	Vrai	Faux	Vrai
Vrai	Faux	Faux	Faux	Vrai
Vrai	Vrai	Faux	Vrai	Vrai

TABLE 3.2 – Table de vérité pour deux variables.

```
chaine1 = 'télé'
chaine2 = 'vision'
chaine3 = chaine1 || chaine2 = 'télévision'
```

3.3.4 Type logique

Une variable de type logique prendra seulement deux valeurs : `Vrai` ou `Faux`. Ce type sera noté `booléen` dans ce cours. Sur des variables de type `booléen`, les opérations suivantes sont possibles :

— La négation : $\neg$ ou `Non`

— La conjonction : `Et` ou $\wedge$

— La disjonction : `Ou` ou $\vee$

Exemple : Soient `A, B` deux variables de type `booléen`, la table 3.2 (de vérité) montre le résultat de chacune des trois opérations vues précédemment.

Notons que les opérateurs de comparaison entre les différents types ont le type logique (`booléen`). Par exemple, pour deux variables `A,B` (de type `entier` ou n'importe quel autre type préalablement défini), la comparaison `A > B` aura le résultat `Vrai` ou `Faux`.

3.4 Actions de base

Une action en algorithmique est appelée aussi instruction. Dans un algorithme on distingue quatre types d'instruction simples et d'autres composées.

Nous pourrons parler de façon générale d'une action (comme mettre des ingrédients dans une casserole pour une recette de cuisine) ou d'instruction quand il s'agit d'un problème qui sera résolu par une machine informatique. Nous allons dans la suite de ce support parler beaucoup plus d'instructions que d'actions.

Les instructions simples (au nombre de quatre) sont les suivante :

1. Déclaration.
2. Affectation.
3. Lecture.
4. Écriture.

Nous allons dans les sections suivantes détailler chacune de ces quatre instructions simples.

3.4.1 Déclaration

Une déclaration consiste à donner un **identificateur** (ou un nom) à une variable. Elle doit aussi fournir **un type** (ou une forme) pour cette même variable.

Un identificateur (qui représentera l'étiquette de cette variable) doit être donc unique pour chaque variable de notre problème.

3.4.1.1 L'identificateur

Cet identificateur ou nom d'une variable doit obéir à un ensemble de règles suivantes :

— Il doit commencer par une lettre de l'alphabet obligatoirement.

— Il ne doit contenir aucun séparateur (l'espace, la tabulation, + , - , %, ...) sauf le caractère souligné (_).

— Il ne doit pas contenir de lettres accentuées ni de ligatures : (é, è, ç, ô, œ, à,...).

— Il ne doit pas être un mots clé.

— Il doit refléter le rôle joué par par cette variable dans le problème.

3.4.1.2 Le type d'un identificateur

Le type d'une variable indique son genre et doit être un des types prédéfinis précédemment, à savoir, `entier, réel, booléen, car, chaine`.

3.4.1.3 Syntaxe d'écriture d'un identificateur

On parle de syntaxe pour dire façon générale pour écrire une phrase ou un mot. Dans l'algorithmique, nous avons aussi une syntaxe à respecter pour écrire nos algorithmes. Nous avons déjà vu un exemple dans le schéma type d'un algorithme (cf. algorithme 1).

La syntaxe de déclaration d'une variable (quelque soit son type) est la suivante :

var identificateur : **type** ;
liste_identificateurs : **type** ;

On pourra définir des variables de cette façon, par exemple :

var Temperature : **réel** ;
i,j,k : **entier** ;

3.4.2 Affectation

Une affectation consiste a donner à une variable une valeur correspondante. Cette opération consiste donc à copier en mémoire la valeur dans la variable qui se situe à gauche.

3.4.2.1 Syntaxe

On écrit de façon générale une affectation de cette manière : l'expression

identificateur ← valeur ;
identificateur ← expression ;

c'est une opération entre des valeurs ou même des variables. Voici un exemple illustratif de quelques opérations d'affectation :

Temperature ← 37.5 ;
Taux ← (Prix*19)/100 ;
i ← i+1;
Symbole ←'%' ;
Prenom ← "Mehdi";
Etat ← `Vrai`;

Remarque :

— Une affectation est une instruction qui **écrase** le contenu de la variable, i.e., la variable aura la nouvelle valeur en mémoire.

— L'ordre d'exécution des instructions dans un algorithme est très important.

Problème 1. A partir de l'algorithme 2 suivant :

1. Donnez la valeur de chacune des trois variables après chaque instruction de cet algorithme.
2. Que fait cet algorithme.

```
Algorithme : Exécution ;
var X,Y,Z : entier ;
Début
    X ← 3;
    Y ← 5;
    Z ← X;
    X ← Y;
    Y ← Z;
Fin.
```

Algorithme 2 : Ordre d'exécution dans un algorithme.

3.4.3 Lecture

La lecture consiste à introduire une valeur à une variable à partir d'une unité d'entrée (clavier).

3.4.3.1 Syntaxe

Pour effectuer une lecture, on utilise la syntaxe suivante :

```
lire(identificateur);
lire(liste_identificateurs);
```

Voici quelques exemple de lectures :

```
lire(Temperature);
lire(i,j,k);
```

3.4.4 Écriture

Cette instruction permet d'**afficher** la valeur d'une variable ou d'un message sur une unité de sortie (écran).

Affichage d'un message :
`ecrire("message");`
Affichage du contenu d'une variable :
`ecrire(identificateur);`

Voici un exemple illustratif :

A ← 5;
`ecrire("La valeur de A =");`
`ecrire(A);`

3.5 Actions composées et expressions

Nous allons dans cette partie parler des actions composées et des expressions qui vont nous servir ensuite pour concevoir la plus part de nos algorithmes.

3.5.1 Actions composées

Une action composée est une suite d'actions simples. On considère donc tout algorithme comme action(s) composée(s) pour résoudre un problème donné. Voici un exemple d'un algorithme simple (algo 3) :

3.5.2 Expressions et opérateurs

Une expression est un ensemble de valeurs reliées par des opérateurs où le résultat final est une valeur. L'ensemble des opérateurs appliqués à ces valeurs représentera le type de cette expression. On aura des expressions qu'on va appeler algébriques pour tout les types qu'on a vu dans ce cours sauf le dernier (booléen). Pour le type logique on définit des expressions logiques. Exemple :

— `25 div 3 mod 2`
— `-25+(3-6)*2+5`

Algorithme : Exemple
var a,b : **réel** ;
Début
    ecrire("Donnez la valeur de a");
    lire(a);
    ecrire("Donnez la valeur de b");
    lire(b);
    a ← a*b+a/2;
    ecrire("a = ",a);
Fin.

Algorithme 3 : Action composée.

Opérateurs	Hiérarchie
( )	1
^	2
− (signe moins)	3
*, /, div, mod	4
+, -	5

TABLE 3.3 – Hiérarchie des opérateurs dans les expressions algebriques.

— `a*5+3*2/b`
— `A Et B Ou Non(C Et D)`
— `"Bon" || "jour" || C`

3.5.3 Hiérarchie des opérations

L'ordre d'exécution des opération dans une expression est très important. Une expression peut parfois, si on ne connais pas les priorités entre les opérations, donner de fausses résultats. Exemple :
`25*3+6` aura le résultat 225 si on exécute l'opération `+` en premier. Or le bon résultat est bien 81.

Les tableaux 3.3 et 3.4 nous montrent la priorité entre les différentes opérateurs dans les expressions :

Pour évaluer donc une expression, on procède de gauche à droite en respectant les différentes priorités définies dans les tableaux cités précédemment.

Opérateurs	Hiérarchie
`( )`	1
`Non`	2
`Et, Ou`	3

TABLE 3.4 – Hiérarchie des opérateurs dans les expressions logiques.

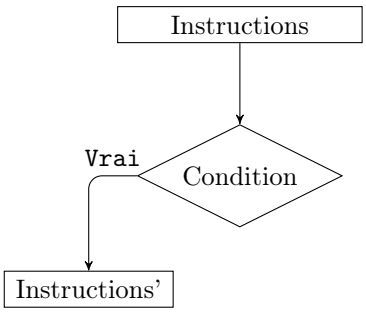

FIGURE 3.2 – Organigramme pour la première forme d'un **si**.

3.6 Structures alternatives

Les structures alternatives ont deux syntaxes (ou formes) différentes. Il s'agit dans la première forme de faire un traitement si une condition est vérifiée et rien dans le cas contraire. Quant à la deuxième forme, on a à exécuter un traitement si une condition est satisfaite et un autre traitement si elle ne l'est pas.

3.6.1 Première forme d'une condition

On exprime très souvent les étapes nécessaires (ou les instructions) d'un algorithme sous forme d'organigramme, i-e., un ensemble de forme géométrique dans un schéma pour représenter la résolution d'un problème. Pour les structures alternatives, on utilise un losange tel que montré dans la figure 3.2. Voici la première forme d'une condition dans un algorithme :

si Condition **alors**
  instructions' ;
finsi;

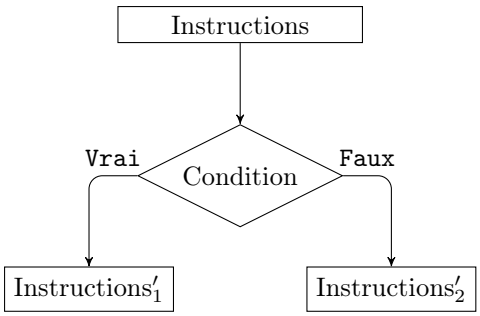

FIGURE 3.3 – Organigramme pour la seconde forme d'un **si**.

Maintenant que nous avons cette instruction nous pouvons écrire correctement l'exercice de la division. Voici la solution (cf. algo 4) :

```
Algorithme : Division2 ;
var x, y, D : réel ;
Début
    ecrire("Donnez la valeur de x");
    lire(x);
    ecrire("Donnez la valeur de y");
    lire(y);
    si y ≠ 0 alors
        D ← x/y;
        ecrire("D = ",D);
    finsi;
Fin.
```

Algorithme 4 : Division de deux nombres réels.

3.6.2 Deuxième forme d'une condition

Il s'agit de prévoir deux traitements différents suivant une condition. La figure 3.3 montre une telle situation. La syntaxe de cette forme est la suivante :

Problème 1. Écrivez un algorithme qui détermine si un nombre entier positif

```
si Condition alors
    instructions'_1 ;
sinon
    instructions'_2;
finsi;
```

(n) donné par l'utilisateur est pair ou impaire :

$$n : \begin{cases} \text{si } \exists\, k \in \mathbb{N},\ n = 2k, & \text{pair,} \\ \text{sinon,} & \text{impair.} \end{cases}$$

Solution : L'algorithme 5 vérifie cette propriété pour tout nombre n entier :

```
Algorithme : Parity ;
var n : entier ;
Début
    ecrire("Donnez la valeur de n");
    lire(n);
    si n mod 2 = 0 alors
        ecrire(n, "est un nombre paire");
    sinon
        ecrire(n, "est un nombre impaire");
    finsi;
Fin.
```

Algorithme 5 : Parité d'un nombre entier.

3.6.3 La condition composée

Ce que nous avons vu jusqu'à maintenant concerne simplement les conditions simples. Dans cette section nous allons aborder les conditions composées.

Une condition composée est construite à partir d'une combinaison (à l'aide des opérateurs logiques) de deux ou plusieurs conditions simples. Par exemple si on veut savoir si un nombre $x \in \mathbb{R}$ appartient à un intervalle défini ainsi : $[-2, 0]$, on aura à vérifier si $x \geq -2$ **et** $x \leq 0$. On écrira donc ce qui suit :

A partir des opérateurs logiques (Et, Ou et Non) et des opérateurs de comparaisons ($<$, $>$, etc), nous pourrions donc construire nos conditions composées.

si x ≥ -2 **Et** x ≤ 0 **alors**
  `ecrire`(”x est dans l’intervalle souhaité”);
sinon
  `ecrire`(”x n’est pas dans l’intervalle souhaité”);
finsi;

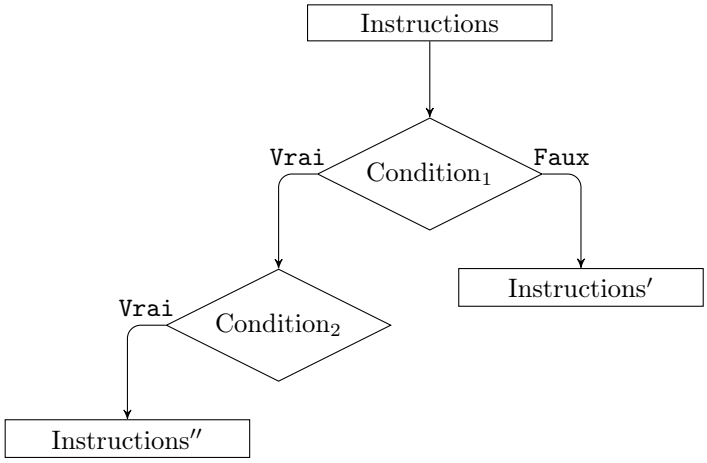

FIGURE 3.4 – Imbrication de deux structures de contrôle.

3.6.4 Les conditions imbriquées

On a vu dans les sections précédentes que dans la structure de condition, on construit des instructions. Puisque, la condition est elle même une instruction, on pourra donc avoir une condition à l’intérieur d’une autre. C’est ce qu’on appelle l’imbrication des structures alternatives. Il est parfois primordial d’utiliser cette imbrication pour résoudre un certain type de problème. L’organigramme 3.4 montre une imbrication de deux conditions de façon générale.

Nous allons essayer d’appliquer ça dans un problème courant.

Problème 1. Il s’agit de résoudre l’équation 3.1 du second degré suivante :

$$ax^2 + bx + c = 0, \quad a, b, c \in \mathbb{R} \tag{3.1}$$

Pour que cette équation soit du second degré, on suppose que $a \neq 0$. Nous allons recevoir les valeurs des variables a, b et c par le biais du clavier de l’utilisateur.

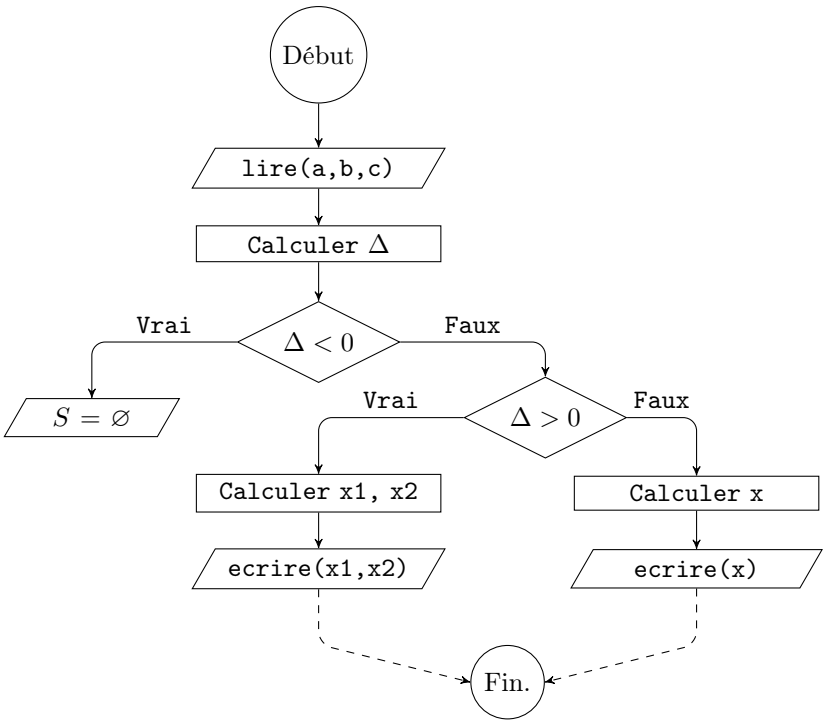

FIGURE 3.5 – Organigramme de la solution d'une équation du second degré ($a \neq 0$).

Solution : Nous savons que l'ensemble des solutions de cette équation dans $\mathbb{R}$ est le suivant :

1. Aucune solution si son discriminant $\Delta < 0$. On dira que $S = \varnothing$, l'ensemble des solutions est vide.
2. Une solution double si $\Delta = 0$, i.e, $S = \{x\}, \quad x = \dfrac{-b}{2a}$.
3. Sinon deux solutions distinctes : $S = \{x_1, x_2\}, \quad x_{1,2} = \dfrac{-b \mp \sqrt{\Delta}}{2a}$.

A partir de ces cas, nous allons réaliser un organigramme pour résoudre n'importe quelle équation du second degré, dont les valeurs a, b et c seront introduites par un utilisateur. A l'étudiant ensuite, la tâche de traduire l'organigramme 3.5 en un algorithme.

3.6.5 La structure de contrôle Selon

Nous savons maintenant qu'à partir de l'imbrication et la composition des conditions d'une structure alternative, on peut répondre à n'importe quelle problématique de choix. Une alternative offre le choix entre deux valeurs possibles, et si le choix se fera entre un ensemble de valeurs préalablement connus ? L'instruction **Selon** répond à ce type de questionnement. Regardons d'abord, la syntaxe de cette instruction :

selon expression **vaut**
  cas valeur 1 : instructions 1;
  cas valeur 2 : instruction 2;
  ⋮
  cas valeur n : instructions n;
  autres instruction 3;
finselon;

L'expression doit avoir soit un type entier ou caractère. La partie **autres** de cette instruction est optionnelle est peut donc être négligée. Nous allons essayer de résoudre un problème qui fait appel à cette structure.

Problème 1. On souhaite dire pour un nombre entier i donné par l'utilisateur, s'il est de la forme suivante :

$$i : \begin{cases} 3k, & \text{ou bien} \\ 3k+1, & \text{ou bien} \\ 3k+2 \end{cases} \tag{3.2}$$

Écrivez un algorithme qui vérifie cette propriété.

Solution :

3.7 Déroulement d'un algorithme

Le déroulement d'un algorithme est l'exécution des instruction de cet algorithme, en prenant en considération l'état en mémoire (les valeurs) de toutes ces variable, ainsi, que toutes ses sorties (affichages) sur l'écran. Cette tâche est nécessaire pour savoir si notre algorithme exécute bien ces instructions et résout en conséquence, le problème pour lequel il est écrit. On parle en informatique d'un

```
Algorithme : Multiple_de3 ;
var i : entier ;
Début
    ecrire("Donnez la valeur de i");
    lire(i);
    selon (i mod 3) vaut
        cas 0 : ecrire("3k");
        cas 1 : ecrire("3k+1");
        cas 2 : ecrire("3k+2");
    finselon;
Fin.
```

Algorithme 6 : Algorithme des classes d'équivalence de 3.

déroulement en mémoire et sur écran d'un algorithme. Cette partie interviendra tout le temps après avoir écrit l'algorithme pour s'assurer de sa correction. On dira que c'est une étape nécessaire pour répondre affirmativement que notre algorithme se termine, et rend les résultats attendues à la fin de son exécution.

Nous allons faire un déroulement de deux problèmes pour voir en détail comment se fait une telle tâche. Il s'agit des exercices 9 et 8.

Problème 2.

1. Déroulez en mémoire et sur écran l'algorithme 15 du chapitre 3 précédant.
2. Déroulez en mémoire et sur écran l'algorithme 4 du chapitre **??** pour les valeurs de x = 4 et y = 5.
3. Même question pour les de x = 12 et y = 0.

Solution : Pour tous les exercices de déroulement, on construit une table qui contiendra toutes les variables de notre algorithme, toutes les conditions (à l'intérieure d'une instruction **si**) ainsi que l'écran.

1. Question 1 : Dans cet algorithme nous n'avons aucune instruction d'écriture, nous aurons donc une case vide dans la partie écran.

Instr.	A	B	C	Écran
1	2			
2	4			
3	4	12		
4	4	12	4	
5	4	12	8	
6	4	12	-4	
7	-32	12	-4	
8	140	12	-4	
9	140	11	-4	

2. Question 2 : Nous allons maintenant négliger les valeurs précédente et transcrire sur la table d'exécution que les valeurs modifiées des variables.

x	y	D	y ≠ 0	Écran
				Donnez la valeur de x
4				
				Donnez la valeur de y
	5			
			Vrai	
		0.8		
				D=0.8

3. Question 3 : x = 12 et y = 0

x	y	D	y ≠ 0	Écran
				Donnez la valeur de x
12				
				Donnez la valeur de y
	0			
			Faux	

3.8 Conclusion

Nous avons jusqu'ici, les étapes préliminaires pour écrire un algorithme proprement. La conception des algorithmes c'est une étape importante dans tout

processus de résolution de problèmes. L'algorithme final repose sur cette conception et des erreurs dans l'analyse des besoins ou de cette même conception faussera ensuite l'algorithme. Nous allons voir dans les chapitres suivants tous les ingrédients nécessaires à l'écriture des algorithmes.

Chapitre 4

Les variables indicées

Sommaire

4.1 Introduction

Nous avons jusqu'ici vu des notions de variables simples (les cinq types vus précédemment). Il existe un type de variable plus complexe, parmi eux, le type tableau. Ce type permet de regrouper un ensemble de variable de même type dans une seule variable qu'on appellera **tableau** de ce type.

Avant de parler de ces types de variables, les notions de structures itératives s'avèrent primordiales. Les sections suivantes détailleront ces instructions avec des exemples illustratifs.

4.2 Les structures itératives

Nous allons parler maintenant des structures itératives. Le mot itérative nous fait penser directement à des répétitions. Pour que ce concept soit plus clair, nous allons essayer de résoudre un problème qui fait apparaître ce besoin de répétition sur un exemple simple.

Une structure itérative servira donc de répéter (un nombre de fois, ou suivant certaines conditions) un ensemble d'instructions. Ce nombre de fois de répétition va subdiviser les structures itératives en deux classes :

1. Structures itératives à nombre de répétition prédéfini :(**pour**).
2. Structures itératives à nombre de répétition non prédéfini : (**répéter**, **tant que**).

4.2.1 La structure itérative pour

Dans cette instruction on connait préalablement le nombre de répétition d'une (ou de plusieurs) instruction(s) donnée(s). La syntaxe générale d'une telle instruction est la suivante :

pour (compteur **de** valeur initiale **à** valeur finale, pas) **faire**
    instructions;
    ⋮
finpour;

Ainsi nous allons exécuter les instructions à l'intérieure de cet algorithme un nombre de fois (nombre répétition) calculé ainsi :

$$\text{nombre répétition} = \frac{\text{valeur finale} - \text{valeur initiale} + 1}{\text{pas}}, \quad \text{pas} \neq 0. \tag{4.1}$$

Le compteur doit être une variable de type entier. De plus, si le pas est un nombre positif, on parle d'une incrémentation de la valeur du compteur, et d'une décrémentation dans le cas contraire. Ce pas ne doit jamais être nul.

Pour bien cerner ce concept, nous allons essayer de résoudre le problème 1 suivant :

Problème 1. On souhaite écrire sur l'écran de l'utilisateur le message "Bonjour !" 10 fois. La solution évidente, serait d'écrire 10 instructions `ecrire()` au sein de notre algorithme. Le but de cet exercice est d'expérimenter l'instruction **pour**. Ce nombre de répétition pourrait être 100 par exemple. Écrivez un algorithme qui fait cette tâche en utilisant la boucle **pour**.

Solution : Nous aurons besoin d'une variable de type entier qui nous servira de compteur de la boucle. En informatique, on a l'habitude d'utiliser les lettres : (i, j, k, ...). Pour respecter cette tradition nous allons déclarer une variable i de type entier. Elle aura comme valeur initiale 1, 10 comme valeur finale et un pas = 1. L'algorithme 7 est la solution à ce problème :

```
Algorithme : Exemple_pour ;
var i : entier ;
Début
    pour (i de 1 à 10,1) faire
        ecrire("Bonjour !");
    finpour;
Fin.
```

Algorithme 7 : Un exemple simple d'instruction pour.

Ce que nous devons retenir de cet exemple simple, et que le compteur i commenceras par la valeur initiale (1) qui sera comparée à la valeur finale (10), si cette condition est **vrai**, l'instruction (`ecrire()`) sera exécutée, le compteur i sera incrémenté du pas (i devient 2), il sera comparé de nouveau avec la valeur finale. L'exécution se termine quand la condition devienne fausse. Dans cet exercice, la condition à vérifier est : (i $\leq$ valeur finale).

Nous allons imaginer un autre problème plus concret, qu'on résoudra et qu'on déroulera, pour illustrer bien les notions de boucles.

Problème 2. On veut calculer la somme des N (donné par l'utilisateur) nombre entier positif. Écrivez un algorithme qui réalise cette tâche.

Solution : Nous avons à calculer une certaine somme qu'on appellera S pour un certain nombre d'entier positif (N). Si N = 4 par exemple, S sera égale à

1+2+3+4. Ainsi de façon générale pour n'importe quel nombre N on aura :

$$S = 1 + 2 + 3 + \cdots + N - 1 + N$$

En notation sigma on aure :

$$S = \sum_{i=1}^{N} i,$$

A partir de ça, nous saurons que nous avons affaire à une boucle pour avec i comme compteur qui commencera par la valeur initiale 1, s'incrémentera par 1, jusqu'à atteindre la valeur N.

```
Algorithme : Somme ;
var i, S, N : entier ;
Début
    ecrire("Donnez la valeur de N");
    lire(N);
    S ← 0;
    pour (i de 1 à N,1) faire
        S ← S + i ;
    finpour;
    ecrire("S = ",S);
Fin.
```

Algorithme 8 : La somme des N premiers entiers.

Nous allons dérouler cet algorithme pour la valeur de N = 3.

i	S	N	i ≤ N	Écran
				Donnez la valeur de N
		3		
	0			
1				
			V	
	1			
2				
			V	
	3			
3				
			V	
	6			
4				
			F	
				S=6

4.2.2 L'instruction répéter

Après avoir vu les boucles pour, nous définissons ici les structures itératives où le nombre d'itérations n'est pas préalablement connu. On répète un ensemble d'instructions jusqu'à ce que certaines conditions soient satisfaites. Il s'agit dans cette situation d'itérer les instructions à l'intérieure de cette boucle *jusqu'à ce que* la Condition soit vraie.

Une syntaxe générale pour cette instruction s'écrit comme suit :

répéter
  instructions;
  ⋮
jusqu'à Condition;

Nous pouvons reproduire le problème 2 et nous essayerons de le résoudre à l'aide de cette instruction. La condition d'arrêt serait qu'un compteur (i), qui nous servira pour compter le nombre d'entier à sommer, sera i > N.

Solution :

```
Algorithme : SommeRepeat ;
var i, S, N : entier ;
Début
    ecrire("Donnez la valeur de N");
    lire(N);
    S ← 0;
    i ← 1;
    répéter
        S ← S + i ;
        i ← i+1;
    jusqu'à (i > N);
    ecrire("S = ",S);
Fin.
```

Algorithme 9 : La somme des N premiers entiers (Répéter).

4.2.3 L'instruction Tant que

Il s'agit dans cette situation d'itérer les instructions à l'intérieure de cette boucle **tant que** la Condition est vraie :

```
tant que Condition faire
    instructions;
    ⋮
fintq;
```

Problème 1. Dites pour chaque nombre saisi ($N \in \mathbb{N}$), s'il est pair ou impair, jusqu'à ce que N soit égale à -1 (l'utilisateur interrompt l'exécution en fournissant la valeur -1).

Solution :

La différence entre ces deux algorithmes est que la boucle **répéter** exécute au moins une fois les instructions à l'intérieur, qui n'est pas le cas de l'instruction **tant que** (sur cet exemple, si N = -1 au début, l'algorithme se terminera).

Nous devons faire attention à nos algorithmes qui comprennent des instructions itératives car on risque d'avoir des boucles infinies (les conditions d'arrêts ne seront jamais satisfaites) !

```
Algorithme : ParityWhile ;
var N : entier ;
Début
    lire(N);
    tant que N ≠ -1 faire
        si N mod 2 = 0 alors
            ecrire(N, " est pair")
        sinon
            ecrire(N, " est impair")
        finsi;
        lire(N);
    fintq;
Fin.
```

Algorithme 10 : Parité des nombres entiers.

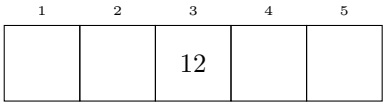

FIGURE 4.1 – Tableau d'entiers contenant 5 valeurs.

4.3 Les tableaux à une seule dimension

Les structures itératives seront nécessaires pour ce type de variable (pour la lecture des éléments d'un tableau par exemple). Ce tableau contiendra donc une seule dimension (des lignes généralement) et un ensemble déterminé de valeur d'un certain type (entier ici). La figure 4.1 nous montre une montre un tableau de 5 valeurs entières avec la valeur 12 dans sa troisième case.

Pour mieux cerner la notion de tableaux en algorithmique, nous allons essayer de résoudre le problème suivant :

Problème 2. Écrivez un algorithme qui calcule la moyenne des étudiants d'une promotion donnée.

Solution : Supposant maintenant que nous voulons pas seulement calculer la moyenne de la promotion, mais aussi sauvegarder la note de chaque étudiant en mémoire ! Comme solution évidente, nous pouvons prévoir une variable pour

```
Algorithme : Moyenne ;
var i, nbrEtud, somme : entier ;
note, moy : réel ;
Début
    ecrire("Donnez le nombre d'étudiants");
    lire(nbrEtud);
    somme ← 0;
    pour i de 1 à nbrEtud,1 faire
        ecrire("Donnez la note de l'étudiant",i);
        lire(note);
        somme ← somme + note;
    finpour;
    moy ← somme / nbrEtud;
    ecrire("La moyenne =",moy);
Fin.
```

Algorithme 11 : La moyenne d'une promotion.

chaque étudiant. Cette solution bien qu'elle répond à notre question mais notre algorithme ne deviendra plus lisible, du moment qu'avec un nombre d'étudiant de 100, nous aurons 100 variables à déclarer ! Nous nous perdrons sûrement pour retrouver nos notes. Les tableaux ont pour rôle principal de sauvegarder un ensemble de valeurs en mémoire sous un même nom qui sera l'identificateur de notre tableau.

4.3.1 Syntaxe de déclaration d'un tableau

Nous préciserons ici qu'un tableau représente un type à lui seul. Ce type diffère des autres vus précédemment puisqu'il s'agit d'un ensemble complexe de donnée. Ainsi, pour pouvoir manipuler des tableaux nous allons faire appel à un type particulier de déclaration. Nous allons entre autre définir un nouveau type qu'on baptisera **Tableau** qui va nous servir pour déclarer une collection de donnée de même type. Cette déclaration est donnée comme suit :

```
Type NotreTableau : tableau de N entier ;
```

On définit un nouveau type (`NotreTableau`) qui est un tableau de N valeurs

entières. La valeur de N doit être connu préalablement (une constante) et le type entier peut être remplacé par n'importe quel autre type de donnée.

La déclaration d'un tableau en algorithme consiste en la donnée de son *identificateur*, sa *taille* et le *type* des valeurs qu'il va contenir. Nous allons supposer que la taille d'un tableau est préalablement connu.

Problème 1. Reprenons le problème 2 traité, nous allons essayer maintenant de sauvegarder les notes de chaque étudiant (au nombre de 100) dans une même variable (Notes).

Solution : Nous remarquons sur cet exemple que l'accès (en lecture et en

```
Algorithme : NotesEtudiants ;
Type NotreTableau : tableau de 100 réel ;
var i : entier ;
Notes : NotreTableau;
Début
    pour i de 1 à 100,1 faire
        ecrire("Donnez la note de l'étudiant ",i);
        lire(Notes[i]);
    finpour;
Fin.
```

Algorithme 12 : Un tableau avec les notes d'une promotion.

écriture) à la note d'un étudiant i se fait à l'aide de l'instruction Notes[i] de façon générale.

4.3.2 Opérations sur les tableaux

Après avoir déclarer un tableau, toutes les opérations qu'on peut faire sur un type donné, peuvent être faite sur ce tableau. En d'autre termes sur les valeurs de ce tableau. Supposant qu'on veut ajouter un point pour l'étudiant 26, cela se fait de la manière suivante :

Notes[26] $\leftarrow$ Notes[26] + 1;

Cette dernière opération reste valable pour n'importe quelle autre opération sur les réels. Notes[26] accède à la valeur de la note de l'étudiant numéro 26 dans la liste, à laquelle elle ajoute 1, le résultat est de nouveau affecté à la même note.

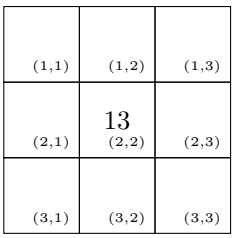

FIGURE 4.2 – Matrice d'entiers de 3×3 valeurs

4.4 Les tableaux à deux dimensions

On appelle parfois un tableau à deux dimensions une matrice. La figure 4.2 nous montre un exemple d'une matrice carrée de 3×3 valeurs entières.

4.4.1 Syntaxe de déclaration d'une matrice

Nous supposons ici qu'on dispose d'un type tableau de N valeurs d'un certain type donnée. Dans cette partie nous allons considérer notre structure `NotreTableau` qui représentera un tableau de 3 valeurs entières. La déclaration qui suit nous donne notre nouveau type `Matrice` :

Type `NotreTableau` : **tableau** de 3 **entier** ;
Type `Matrice` : **tableau** de 3 `NotreTableau` ;

Alors ici, la `Matrice` est un tableau qui contient un autre tableau de 3 entiers. De ce fait, l'accès (en lecture et en écriture) des éléments de cette matrice se fait de la manière suivante :

Supposons qu'on veut accéder à l'élément situé à la deuxième ligne et à la deuxième colonne d'une matrice M (nous nous référons toujours à la matrice de la figure 4.2). Il suffit d'écrire M[2][2].

4.4.2 Opération sur les matrices

Puisque accéder aux éléments de la matrice s'effectue en invoquant l'indice de la ligne et celui de la colonne, nous pouvons facilement faire des opérations sur les éléments de cette matrice de la manière suivant, à titre d'exemple :

M[2][2] ← 12 ;
M[1][1] ← M[2][2] - 3 ;
A ← M[1][1]-M[2][2] ; // A aura la valeur -3 ;
M[1][3] ← M[2][2] **mod** M[1][1] ;
$M[0][0] \leftarrow M[0][0] + 100$;

4.5 Structures itératives imbriquées

Comme on a vu pour les structures alternatives la notion d'imbrication, on peut retrouver cette même notion dans les instructions **pour**, **répéter** et **tant que**. Ce type d'instruction se manifeste beaucoup pour des notions de matrices (des tableaux de tableaux). Contrairement aux tableaux qui ont une seule dimension (un nombre de ligne), ces matrices en ont deux (lignes et colonnes). Nous aurons donc besoins de deux boucles imbriquées pour accéder aux éléments de cette structure.

Pour afficher l'ensemble des valeurs de la matrice vu précédemment (cf. figure 4.2), il faut la parcourir (accéder à l'ensemble des indices donnée ici par les petits nombres entre parenthèses) Il nous faut pour ce faire, parcourir l'ensemble des lignes de 1 à 3 et l'ensemble des colonnes de 1 à 3, on utilise donc l'instruction suivante :

pour i **de** 1 **à** 3,1 **faire**
  pour j **de** 1 **à** 3,1 **faire**
    `lire`(M[i][j]) ; // M est notre matrice
  finpour;
finpour;

Nous invitons nos lecteurs à essayer de résoudre le problème **??** corrigé pour mieux comprendre la manipulation des matrices.

Problème 1. Écrivez un algorithme qui prend en entrée une matrice carrée d'ordre 3 M et qui retourne cette matrice élevée à la puissance 2.

Solution :

```
Algorithme : Matrice ;
Type NotreTableau : tableau de 3 réel ;
Type Matrice : tableau de 3 NotreTableau ;
var M : Matrice;
i,j : entier ;
x : réel ;
Début
    pour i de 1 à 3,1 faire
        pour j de 1 à 3,1 faire
            lire(x);
            M[i][j] ← x*x ;
        finpour;
    finpour;
Fin.
```

Algorithme 13 : Le carré d'une matrice quelconque d'ordre 3.

4.6 Conclusion

Nous avons étudier dans ce chapitres l'ensemble des structures itératives qui servent à répéter un ensemble d'instructions sous certaines conditions. Deux nouveaux types de données ont émergé : Les tableaux. Manipuler des données de ce type nécessite l'utilisation des structures itératives (imbriquées parfois).

Le rôle principal de ces types de données est de sauvegarder en mémoire un ensemble de variables (dont on connait le nombre) de même type. Toutes les opérations connus sur un type (primitif) de données seront ensuite valables sur les tableaux d'une et de deux dimensions. Il suffit de respecter les opérations d'accès aux éléments de ces structures.

Annexe A

Exercices divers

A.1 Codage de l'information

Exercice 1. Décoder les nombres ci-dessous :

$$(1110001)_2 = \cdots \qquad (11111011000)_2 = \cdots \qquad (11111111)_2 = \cdots$$
$$(341)_{16} = \cdots \qquad (11111111)_2 = \cdots \qquad (11010)_{16} = \cdots$$

Exercice 2. Codage :

1. Coder en binaire les nombres ci-dessous :

$$(57)_{10} = \cdots \qquad (168)_{10} = \cdots \qquad (255)_{10} = \cdots$$

2. Coder en hexadécimal les nombres ci-dessous :

$$(2016)_{10} = \cdots \qquad (255)_{10} = \cdots \qquad (400)_{10} = \cdots$$

Exercice 3. Transcodage :

1. Transcoder les nombres ci-dessous en base 2 :

$$(\mathrm{A8})_{16} = \cdots \qquad (\mathrm{C5})_{16} = \cdots \qquad (\mathrm{4FF0})_{16} = \cdots \qquad (806)_{16} = \cdots$$

2. Transcoder les nombres ci-dessous en base 16 :

$$(11100111)_2 = \cdots \quad (1100110011)_2 = \cdots$$
$$(1000011111000)_2 = \cdots \quad (1010101010)_2 = \cdots$$

Exercice 4. Nombre signés :

1. Coder les valeurs suivantes en complément à 2 : 0, 10, -10 and -127.
2. Coder -15 en complément à 2 sur 1 octet.
3. Décoder cette valeur binaire 10100010 représentée en complément à 2.

Exercice 5. Prouvez l'égalité suivante :

$$(\mathrm{b})_{10} = (10)_b, \quad \forall b \in \mathbb{N}, b > 1.$$

A.2 Identificateurs, expression, Si et Déroulement

Exercice 6. A partir des identificateurs suivants, dites quels sont ceux qui sont correctes ? Corrigez les identificateurs invalides ensuite :

A	Azerty	A-zer-ty	Exo-1	Exo_2	Var
F%	A B	Un_id	MAX(A,B)	français	lire
A_	5ème_B6	Nom.2	EX97 1	A_1	Ex17

Exercice 7. Évaluez les différentes expressions suivantes :

1. `3^2+4`
2. `3^(2+4)`
3. `17 mod 10 div 3`
4. `12*3+5`
5. `5.*4.*9.^2`
6. `E Ou F Et G` avec `E=Vrai, F=Faux` et `G=Vrai`.
7. `7+9/3-10*2`

Correction 1. Les évaluations :

1. `13`
2. `729`
3. `1`
4. `31`
5. `1620`
6. `Vrai`
7. `-10`

Exercice 8. Écrivez un algorithme qui calcule la division de deux nombres réels x et y.

Correction 2. Voici l'algorithme 14 qui réalise l'opération de division :

```
Algorithme : Division ;
var x, y, D : réel ;
Début
    ecrire("Donnez la valeur de x");
    lire(x);
    ecrire("Donnez la valeur de y");
    lire(y);
    D ← x/y;
    ecrire("D = ",D);
Fin.
```

Algorithme 14 : Division de deux nombres réels ($y \neq 0$).

Exercice 9. Soit l'algorithme suivant, donnez les valeurs des variables A, B et C après l'exécution de chaque instruction de cet algorithme !

```
Algorithme : PGCD ;
var A, B, C : entier ;
Début
    A ← 2;
    A ← A+2;
    B ← A*2+A;
    C ← 4;
    C ← B-C;
    C ← A-C ;
    A ← (B-A)*C ;
    A ← B+C*A;
    B ← (A+C)/B;
Fin.
```

Algorithme 15 : Déroulement d'un algorithme.

Exercice 10. Utilisez la fonction `mod` pour vérifier la parité d'un nombre entier positif $m \in \mathbb{N}$.

A.3 Structures itératives

Exercice 11. Écrivez un algorithme qui calcule et affiche la somme suivante :

$$S = N + (N-1) + \cdots + 2 + 1$$

Exercice 12. Écrivez un algorithme qui calcule et affiche le miroir de la conversion binaire d'un nombre décimal. Par exemple si on prend N = 13 :

$$(13)_{10} = (1101)_2, \quad \widetilde{(1101)_2} = (1011)_2.$$

Ainsi notre algorithme nous fournira pour N = 13, la sortie $(1011)_2$

Exercice 13. Écrivez l'algorithme qui détermine pour toute pair de nombres entiers (a, b), saisies au clavier, le plus grand diviseur commun, `PGCD(a,b)`.

Correction 3. Voici l'algorithme d'*Euclide* :

```
Algorithme : PGCD ;
var A, B, r : entier ;
Début
    lire(A,B);
    tant que (B != 0) faire
        r ← A mod B ;
        A ← B ;
        B ← r ;
    fintq;
    ecrire("PGCD = ",A);
Fin.
```

Exercice 14. Écrivez un algorithme qui calcule une approximation de e donnée par la formule suivante :

$$e = 1 + \frac{1}{2} + \frac{1}{6} + \frac{1}{24} + \cdots + \frac{1}{N(N-1)\cdots 1}$$

Indications 1.

$$e = \sum_{i=0}^{n} \frac{1}{i!}$$

Exercice 15. Écrivez un algorithme qui calcule une approximation de $\frac{\pi}{2}$ donnée par la formule suivante :

$$\frac{\pi}{2} \simeq \left(\frac{2}{1}\frac{2}{3}\right)\left(\frac{4}{3}\frac{4}{5}\right)\cdots\left(\frac{20}{19}\frac{20}{21}\right)$$

Indications 2. Il suffit de voir que :

$$\frac{\pi}{2} \simeq \prod_{k=1}^{N} \frac{2k}{2k-1}\frac{2k}{2k+1}$$

Exercice 16. Écrivez un algorithme qui calcule une approximation de $\frac{\pi}{4}$ donnée par la formule suivante :

$$\frac{\pi}{4} \simeq 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots$$

Indications 3. On peut écrire :

$$\frac{\pi}{4} \simeq \sum_{i=0}^{n} \frac{(-1)^i}{2i+1}$$

Exercice 17. Écrivez un algorithme qui calcule une approximation de $\cos(x)$ pour chaque $x \in \mathbb{R}$ donné, à l'aide de la formule suivante :

$$\cos(x) \simeq 1 - \frac{x^2}{2!} + \frac{x^4}{4!} + \cdots + (-1)^n \frac{x^{2n}}{(2n)!}$$

Indications 4. On peut écrire :

$$\cos(x) \simeq \sum_{i=0}^{n} (-1)^i \frac{x^{2i}}{(2i)!}$$