Initiation à l'algorithmique

Cours

Public Concerné : Licence Sciences Cognitives
Origine : adaptation du cours de 1ère année IUT Charlemagne (Y. Belaïd et B. Girau)

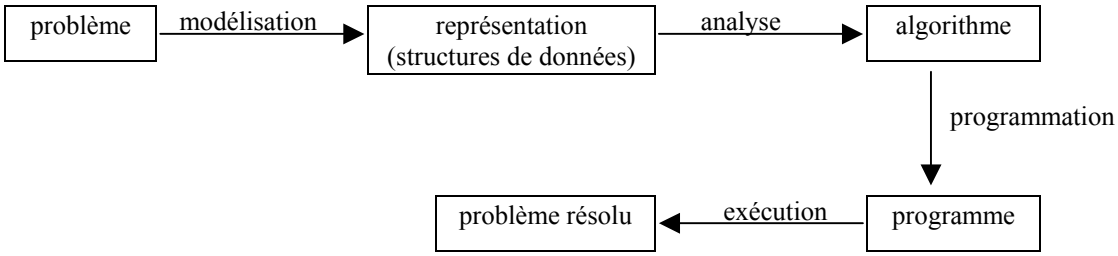
NOM DE L'AUTEUR : A. Belaïd **DATE : 2003**

UNIVERSITE NANCY 2
Département de Math-Info
54000 NANCY cedex
Abdel.Belaid@univ-nancy2.fr

1 Instructions et types élémentaires

1.1 Introduction à l'algorithmique

Un algorithme est une étape préalable à l'écriture d'un programme informatique. Il décrit le résultat de l'analyse d'un problème (énoncé en français) dans un langage formel. Il se présente sous la forme d'une liste d'opérations permettant de réaliser un travail. L'ordre de ces opérations et leur enchaînement est très important. La figure suivante décrit les phases nécessaires à la résolution d'un problème.



Le « langage algorithmique» utilisé est un compromis entre un langage naturel et un langage de programmation. Un algorithme est une suite d'instructions présentées dans l'ordre des traitements. Les instructions sont des opérations composées de variables, de constantes et d'opérateurs. L'algorithme sera toujours accompagné d'un lexique qui indique, pour chaque variable, son type et son rôle. Un algorithme est délimité par les mots clés *début* et *fin*. Nous manipulerons les types couramment rencontrés dans les langages de programmation : *entier*, *réel*, *booléen*, *caractère*, *chaîne*, *tableau* et *type composite*.

1.2 Notions de variables, types et valeurs

Les variables d'un algorithme permettent de contenir les informations intermédiaires pour établir un raisonnement. Chaque variable a un nom (identifiant) et un type. Ce dernier correspond au genre d'information que l'on souhaite utiliser :

- **entier** pour manipuler des nombres entiers,
- **réel** pour manipuler des nombres réels,
- **booléen** pour manipuler des valeurs booléennes **vrai** ou **faux**,
- **caractère** pour manipuler des caractères alphabétiques et numériques,
- **chaîne** pour manipuler des chaînes de caractères permettant de représenter des mots ou des phrases.

Il faut noter qu'à un type donné, correspond un ensemble d'opérations définies pour ce type. Une variable est l'association d'un nom avec un type, permettant de mémoriser une valeur de ce type.

Le type entier

Les opérations utilisables sur les entiers sont :

- les opérateurs arithmétiques classiques : + (addition), - (soustraction), * (produit)
- la division entière, notée ÷, telle que $n \div p$ donne la partie entière du quotient de la division de n par p
- le modulo, notée **mod**, telle que $n \bmod p$ donne le reste de la division entière de n par p
- les opérateurs de comparaison classiques : <, >,=,≠, ≥, ≤

Le type réel

Les opérations utilisables sur les réels sont :

- les opérateurs arithmétiques classiques : + (addition), - (soustraction), * (produit), / (division)
- les opérateurs de comparaison classiques : <, >,=,≠, ≥, ≤

Le type booléen

Il s'agit du domaine dont les seules valeurs sont *vrai* ou *faux*. Les opérations utilisables sur les booléens sont réalisées à l'aide des connecteurs logiques : et (pour le *et* logique), ou (pour le *ou* logique), non (pour le *non* logique).

Rappel :

non	
vrai	faux
faux	vrai

et	*vrai*	*faux*
vrai	vrai	faux
faux	faux	faux

ou	*vrai*	*faux*
vrai	vrai	vrai
faux	vrai	faux

Le type caractère

Il s'agit du domaine constitué des caractères alphabétiques et numériques. Les opérations élémentaires réalisables sont les comparaisons : <, >,=,≠, ≥, ≤.

Le type chaîne

Une chaîne est une séquence de plusieurs caractères. Les opérations élémentaires réalisables sont les comparaisons : <, >,=,≠, ≥, ≤ selon l'ordre lexicographique.

1.3 Instructions d'affectation et expressions

Une instruction traduit une ou plusieurs actions portant sur une ou plusieurs variables. Ces actions sont relatives aux opérations admissibles sur les valeurs des variables. L'instruction la plus commune est l'**affectation**. Elle consiste à doter une variable d'une valeur appartenant à son domaine, c'est-à-dire à lui donner une première valeur ou à changer sa valeur courante. Elle se note par le signe ←.

Une expression est une suite finie bien formée d'opérateurs portant sur des variables ou des valeurs et qui a une valeur. La valeur de l'expression doit être conforme au domaine de la variable affectée.

Exemple d'algorithme

algorithme
début
x ← 12
y ←x + 4
x ←3
fin
lexique
- x : entier
- y : entier

On remarque que les deux premières instructions ne sont pas permutables car *x* n'aurait alors pas de valeur au moment du calcul de l'expression.

Schéma de l'état des variables au cours de l'exécution :

Schéma de l'évolution de l'état des variables instruction par instruction :

instructions / variables	1	2	3
x	12		3
y		16	

1.4 Instructions de lecture et d'écriture

Instruction de lecture

L'instruction de prise de données sur le périphérique d'entrée (en général le clavier) est :

variable ← **lire** ( )

L'exécution de cette instruction consiste à affecter une valeur à la variable en prenant cette valeur sur le périphérique d'entrée. Avant l'exécution de cette instruction, la variable avait ou n'avait pas de valeur. Après, elle a la valeur prise sur le périphérique d'entrée.

Instruction d'écriture

L'instruction de restitution de résultats sur le périphérique de sortie (en général l'écran) est :

écrire (liste d'expressions)

Cette instruction réalise simplement l'affichage des valeurs des expressions décrites dans la liste. Ces expressions peuvent être simplement des variables ayant des valeurs ou même des nombres ou des commentaires écrits sous forme de chaîne de caractères.

Exemple d'utilisation : écrire (x, y+2, "bonjour")

Exemple d'algorithme

Écrire un algorithme qui décompose une somme d'argents en billets de 100 euros, 50 euros et 10 euros, et de pièces de 2 euros et 1 euro. La somme sera lue au clavier, et les valeurs affichées une par ligne.

Principe :
L'algorithme commence par lire sur l'entrée standard l'entier qui représente la somme d'argent et affecte la valeur à une variable somme. Pour obtenir la décomposition en nombre de billets et de pièces de la somme d'argent , on procède par des divisions successives en conservant chaque fois le reste.

```
algorithme
        début
        somme ← lire ( )                    // 1
        b100 ← somme ÷ 100                  // 2
        r100 ← somme mod 100                // 3
        b50 ← r100 ÷ 50                     // 4
        r50 ← r100 mod 50                   // 5
        b10 ← r50 ÷ 10                      // 6
        r10 ← r50 mod 10                    // 7
        p2 ← r10 ÷ 2                        // 8
        r2 ← r10 mod 2                      // 9
        p1 ← r2                             // 10
        écrire (b100, b50, b10, p2, p1)     // 11
        fin
```

lexique
- somme : entier, la somme d'argent à décomposer
- b100: entier, le nombre de billets de 100 euros
- b50: entier, le nombre de billets de 50 euros
- b10: entier, le nombre de billets de 10 euros
- p2: entier, le nombre de pièces de 2 euros
- p1: entier, le nombre de pièces de 1 euro
- r100: entier, reste de la division entière de somme par 100
- r50: entier, reste de la division entière de r100 par 50
- r10: entier, reste de la division entière de r50 par 10
- r2: entier, reste de la division entière de r10 par 2

Schéma de l'évolution de l'état des variables instruction par instruction :

instructions / variables	1	2	3	4	5	6	7	8	9	10	11
somme	988										
b100		9									écrire
b50				1							écrire
b10						3					écrire
p2								4			écrire
p1										0	écrire
r100			88								
r50					38						
r10							8				
r2									0		

1.5 Notion de fonction

Une fonction est un algorithme autonome, réalisant une tâche précise. Il reçoit des paramètres (valeurs) en entrée et retourne une valeur en sortie (résultat). La notion de fonction est très intéressante car elle permet, pour résoudre un problème, d'employer une méthode de décomposition en sous-problèmes distincts. Elle facilite aussi la réutilisation d'algorithmes déjà développés par ailleurs.

Une fonction est introduite par un *en-tête*, appelé aussi *signature* ou *prototype*, qui spécifie :
- le nom de la fonction,
- les paramètres donnés et leur type,
- le type du résultat.

La syntaxe retenue pour l'en-tête est la suivante :

fonction nomFonction (liste des paramètres) : type du résultat

La liste des paramètres précise, pour chaque paramètre, son nom et son type. La dernière instruction de la fonction indique la valeur retournée, nous la noterons:

retourne expression

Exemple de fonction

Ecrire une fonction calculant le périmètre d'un rectangle dont on donne la longueur et la largeur.

```
fonction calculerPérimètreRectangle (longueur: réel, largeur: réel) : réel
    début
        périmètre ← 2*(longueur + largeur)
        retourne périmètre
    fin
lexique
- longueur : réel, longueur du rectangle
- largeur : réel, largeur du rectangle
- périmètre : réel, périmètre du rectangle
```

1.6 Instructions conditionnelles

Les exemples précédents montrent des algorithmes dont les instructions doivent s'exécuter dans l'ordre, de la première à la dernière. Nous allons introduire une instruction précisant que le déroulement ne sera plus séquentiel. Cette instruction est appelée une **conditionnelle**. Il s'agit de représenter une alternative où, selon les cas, un bloc d'instructions est exécuté plutôt qu'un autre. La syntaxe de cette instruction est :

```
si condition
        alors liste d'instructions
        sinon liste d'instructions
fsi
```

Cette instruction est composée de trois parties distinctes : la condition introduite par *si*, la clause *alors* et la clause *sinon*. La condition est une expression dont la valeur est de type booléen. Elle est évaluée. Si elle est vraie, les instructions de la clause *alors* sont exécutées. Dans le cas contraire, les instructions de la clause *sinon* sont exécutées.

On peut utiliser une forme simplifiée de la conditionnelle, sans clause *sinon*. La syntaxe est alors :

```
si condition
        alors liste d'instructions
fsi
```

Exemple 1 (conditionnelle simple)

Ecrire un algorithme qui permet d'imprimer le résultat d'un étudiant à un module sachant que ce module est sanctionné par une note d'oral de coefficient 1 et une note d'écrit de coefficient 2. La moyenne obtenue doit être supérieure ou égale à 10 pour valider le module.

données : la note d'oral et la note d'écrit

résultat : impression du résultat pour le module (*reçu* ou *refusé*)

principe : on calcule la moyenne et on la compare à 10.

```
algorithme :
    début
    ne, no ← lire ( )
    moy← (ne * 2 + no) / 3
    si moy ≥ 10
        alors   écrire ("reçu")
        sinon   écrire ("refusé")
    fsi
    fin
lexique :
-   moy : réel, moyenne
-   ne : réel,  note d'écrit
-   no : réel,  note d'oral
```

Exemple 2 (conditionnelles imbriquées)

On veut écrire une fonction permettant de calculer le salaire d'un employé payé à l'heure à partir de son salaire horaire et du nombre d'heures de travail.

Les règles de calcul sont les suivantes : le taux horaire est majoré pour les heures supplémentaires : 25% au-delà de 160 h et 50% au-delà de 200 h.

```
fonction calculerSalaire ( sh : réel, nbh : entier) : réel
    début
    si  nbh ≤ 160
        alors   salaire ← nbh * sh
        sinon   si  nbh ≤ 200
                        alors salaire ← 160 * sh + (nbh - 160) * sh * 1.25
                        sinon salaire ←160 * sh + 40 * sh * 1,25 + (nbh - 200) * sh * 1.5
                fsi
    fsi
```

retourne salaire
fin

lexique
- sh : réel, salaire horaire de l'employé
- nbh : entier, nombre d'heures de travail de l'employé
- salaire : réel, salaire de l'employé

2 Itérations

Il arrive souvent dans un algorithme que la même action soit répétée plusieurs fois, avec éventuellement quelques variations dans les paramètres qui précisent le déroulement de l'action. Il est alors fastidieux d'écrire un algorithme qui contient de nombreuses fois la même instruction. De plus, ce nombre peut dépendre du déroulement de l'algorithme. Il est alors impossible de savoir à l'avance combien de fois la même instruction doit être décrite. Pour gérer ces cas, on fait appel à des instructions en boucle qui ont pour effet de répéter plusieurs fois une même action. Deux formes existent : la première, si le nombre de répétitions est connu avant l'exécution de l'instruction de répétition, la seconde s'il n'est pas connu. On appellera itération l'exécution de la liste des instructions.

2.1 Répétitions inconditionnelles

Il est fréquent que le nombre de répétitions soit connu à l'avance, et que l'on ait besoin d'utiliser le numéro de l'itération afin d'effectuer des calculs ou des tests. Le mécanisme permettant cela est la boucle **Pour**.

Forme de la *boucle Pour* :

Pour variable de valeur initiale à valeur finale faire
    liste d'instructions
fpour

La variable dont on donne le nom va prendre successivement toutes les valeurs entières entre *valeur initiale* et *valeur finale*. Pour chaque valeur prise par la variable, la liste des instructions est exécutée. La variable utilisée pour énumérer les itérations est appelée *variable d'itération*, *indice d'itération* ou *compteur*. L'incrémentation par 1 de la variable est implicite.

Autre forme de la *boucle Pour* :

Pour variable décroissant de valeur initiale à valeur finale faire
    liste d'instructions
fpour

La variable d'itération est décrémentée de 1 après chaque itération.

Exemple 1 (cas simple, compteur croissant)
Ecrire l'algorithme permettant d'afficher la table de multiplication par 9.

Un algorithme possible est le suivant :

```
algorithme
    début
    écrire(1*9)
    écrire(2*9)
    écrire(3*9)
    écrire(4*9)
    écrire(5*9)
    écrire(6*9)
    écrire(7*9)
    écrire(8*9)
    écrire(9*9)
    écrire(10*9)
    fin
```

Il est plus simple d'utiliser une boucle avec un compteur prenant d'abord la valeur 1, puis augmentant peu à peu jusqu'à atteindre 10.

```
algorithme
        début
        pour i de 1 à 10 faire
            écrire (i*9)
        fpour
        fin
lexique
-   i : entier, indice d'itération
```

Exemple 2 (compteur décroissant)

Compte à rebours : écrire l'algorithme de la fonction qui, à partir d'un nombre entier positif *n* , affiche tous les nombres par ordre décroissant jusqu'à 0.

Exemple : pour n = 5, le résultat sera 5 4 3 2 1 0.

```
fonction compteARebours ( n : entier)
        début
        pour i décroissant de n à 0 faire
                écrire (i)
        fpour
        fin
lexique
-   n : entier,
-   i : entier, indice d'itération
```

Exemple 3 (calcul par récurrence : cas de la somme)

On veut imprimer, pour *n* donné, la somme des carrés des *n* premiers entiers.

Cette somme, notée *s*, est obtenue en calculant le n-ième terme d'une suite définie par récurrence :

$$s_n = s_{n-1} + i^2$$

Algorithmiquement, le calcul d'une telle suite se fait en deux étapes :

- initialisation (ici, $s_0 = 0$),
- répétition de : calcul du $i^{ème}$ terme en fonction du terme d'indice i-1

```
algorithme
        début
        n ← lire ( )                    // 1
        s ← 0                           // 2
        pour i de 1 à n faire           // 3
                s ← s + i²              // 4
        fpour
        écrire (s)                      // 5
        fin
lexique
-   s : entier, somme des carrés des n premiers entiers
-   n : entier,
-   i : entier, indice d'itération
```

Schéma de l'évolution de l'état des variables instruction par instruction :

On suppose que la valeur introduite par l'utilisateur est 4 .

variables / instructions	n	s	i
1	4		
2		0	
3			1
4		1	
3			2
4		5	
3			3
4		14	
3			4
4		30	
3			(fin)
5		écrire	

Exemple 4 (calcul par récurrence d'un maximum, initialisation à un terme artificiel)

Ecrire l'algorithme qui permet d'imprimer le maximum de *n* entiers positifs donnés au fur et à mesure.

Comment trouver ce maximum ? C'est le plus grand des 2 nombres : maximum des n-1 premiers entiers positifs donnés, n-ème entier donné. Ceci est une définition par récurrence :

Si $nombre_n > maximum_{n-1}$ alors $maximum_n \leftarrow nombre_n$ sinon $nombre_n \leftarrow maximum_{n-1}$ fsi

Comment initialiser la suite *maximum* ? Il faut que $maximum_1$ prenne la valeur $nombre_1$. Il suffit alors de choisir une valeur de $maximum_0$ plus petite que tout entier positif, par exemple $maximum_0 = -1$, de manière à assurer que $maximum_1$ aura bien $nombre_1$ pour valeur. Il s'agit d'une initialisation à un terme artificiel.

```
algorithme
        début
        n ← lire ( )                          // 1
        maximum ← -1                          // 2
        pour i de 1 à n faire                 // 3
            nombre ← lire ( )                 // 4
            si nombre > maximum               // 5
                    alors maximum ← nombre    // 6
            fsi
        fpour
        écrire (maximum)                      // 7
        fin
lexique
-   n : entier,
-   maximum : entier, maximum des i premiers nombres entiers
-   nombre : entier, ième entier positif donné
-   i : entier, indice d'itération
```

Schéma de l'évolution de l'état des variables instruction par instruction :

On suppose que les valeurs introduites par l'utilisateur sont : 4 2 0 8 7.

variables / instructions	n	maximum	i	nombre	nombre>maximum
1	4				
2		0			
3			1		
4				2	
5					vrai
6		2			

3			2		
4				0	
5					faux
3			3		
4				8	
5					vrai
6		8			
3			4		
4				7	
5					faux
3			(fin)		
7		écrire			

Exemple 5 (calcul par récurrence d'un maximum, initialisation à un terme utile)

Ecrire l'algorithme qui permet d'imprimer le maximum de *n* entiers donnés.

L'initialisation à un terme artificiel n'est plus possible ici car les valeurs ne sont plus bornées inférieurement. Une solution consiste alors à initialiser au premier terme et à commencer la formule générale de récurrence à 2. Il s'agit d'une initialisation à un terme utile.

```
algorithme
        début
        n ← lire ( )                          // 1
        maximum ← lire ( )                    // 2
        pour i de 2 à n faire                 // 3
            nombre ← lire ( )                 // 4
            si nombre > maximum               // 5
                    alors maximum ← nombre    // 6
            fsi
        fpour
        écrire (maximum)                      // 7
        fin
lexique
- n : entier,
- maximum : entier, maximum des i premiers nombres entiers
- nombre : entier, ième entier positif donné
- i : entier, indice d'itération
```

Schéma de l'évolution de l'état des variables instruction par instruction :

On suppose que les valeurs introduites par l'utilisateur sont : 4 2 0 8 7.

variables / instructions	n	maximum	i	nombre	nombre>maximum
1	4				
2		2			
3			2		
4				0	
5					faux
3			3		
4				8	
5					vrai
6		8			
3			4		
4				7	
5					faux
3			(fin)		
7		écrire			

2.2 Répétitions conditionnelles

L'utilisation d'une boucle *pour* nécessite de connaître à l'avance le nombre d'itérations désiré, c'est-à-dire la valeur finale du compteur. Dans beaucoup de cas, on souhaite répéter une instruction tant qu'une certaine condition est remplie, alors qu'il est a priori impossible de savoir à l'avance au bout de combien d'itérations cette condition cessera d'être satisfaite. Le mécanisme permettant cela est la boucle **Tant que**.

Syntaxe de la *boucle Tant que* :

```
tant que condition faire
        liste d'instructions
ftant
```

Cette instruction a une condition de poursuite dont la valeur est de type booléen et une liste d'instructions qui est répétée si la valeur de la condition de poursuite est vraie : la liste d'instructions est répétée autant de fois que la condition de poursuite a la valeur vraie. Le déroulement pas à pas de cette instruction équivaut à :

```
si condition
        alors   liste d'instructions
                si condition
                        alors   liste d'instructions
                                si condition
                                        alors   liste d'instructions
                                                …
```

Etant donné que la condition est évaluée avant l'exécution des instructions à répéter, il est possible que celles-ci ne soient jamais exécutées. Il faut que la liste des instructions ait une incidence sur la condition afin qu'elle puisse être évaluée à faux et que la boucle se termine. Il faut toujours s'assurer que la condition devient fausse au bout d'un temps fini.

Exemple 1

On veut laisser un utilisateur construire des rectangles de taille quelconque, à condition que les largeurs qu'il indique soient supérieures à 1 pixel. On peut utiliser une répétition conditionnelle qui permet de redemander à l'utilisateur de saisir une nouvelle valeur tant que celle-ci n'est pas valide.

```
fonction saisirLargeurRectangle ( ) : entier
        début
        écrire ("indiquez la largeur du rectangle : ")
        largeur ← lire ( )
        tant que largeur < 1 faire
                écrire ("erreur : indiquez une valeur strictement positive")
                écrire ("indiquez la largeur du rectangle : ")
                largeur ← lire ( )
        ftant
        retourne largeur
        fin
lexique
-   largeur : entier, largeur courante saisie
```

Exemple 2

Un poissonnier sert un client qui a demandé 1kg de poisson. Il pèse successivement différents poissons et s'arrête dès que le poids total égale ou dépasse 1kg. Donner le nombre de poissons servis.

Remarque sur la terminaison :
Ce problème est typique des cas où le dernier terme (celui qui fait basculer le test) doit être retenu. Nous verrons en exercice des problèmes dans lesquels le dernier terme (celui qui fait basculer le test) doit être rejeté (exemple : le passager d'un ascenseur qui fait dépasser la charge maximale).

données
poids des poissons successifs en grammes

résultats
nombre de poissons vendus

algorithme
début
poidstotal ← 0 // 1
nbpoisson ← 0 // 2
tant que poidstotal < 1000 faire // itération i // 3
poidspoisson ← lire ( ) // 4
nbpoisson ← nbpoisson + 1 // 5
poidstotal ← poidstotal + poidspoisson // 6
ftant
écrire (nbpoisson) // 7
fin
lexique
- poidspoisson : réel, poids du i ème poisson, en grammes
- nbpoisson : entier, nombre de poissons vendus après la i ème itération (c'est i)
- poidstotal : réel, poids total après la i ème itération (poids des i premiers poissons)

Schéma de l'évolution de l'état des variables instruction par instruction :

On suppose que les valeurs introduites par l'utilisateur sont : 350 280 375.

variables / instructions	poidstotal	nbpoisson	poidspoisson	poidstotal < 1000
1	0			
2		0		
3				vrai
4			350	
5		1		
6	350			
3				vrai
4			280	
5		2		
6	630			
3				vrai
4			375	
5		3		
6	1005			
3				faux
7		écrire		

2.3 Boucles imbriquées

Le corps d'une boucle est une liste d'instructions. Mais cette boucle est elle-même une instruction. Donc le corps d'une boucle peut contenir une boucle dite *imbriquée*, qui utilise un compteur différent. Il faut alors faire attention à l'ordre d'exécution des instructions : à chaque passage dans le corps de la boucle principale, la boucle imbriquée va être exécutée totalement.

Exemple

Ecrire l'algorithme permettant d'imprimer le triangle suivant, le nombre de lignes étant donné par l'utilisateur :

1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
…

algorithme
début
nblignes ← lire ( ) // 1
pour i de 1 à nblignes faire // 2
pour j de 1 à i faire // 3
écrire (j) // 4
fpour
retour à la ligne //5
fpour
fin

lexique
- nblignes : entier, nombre de lignes à imprimer
- i : entier, indice d'itération sur les lignes
- j : entier, indice d'itération sur les éléments de la i ème ligne

Schéma de l'évolution de l'état des variables instruction par instruction :

On suppose que la valeurs introduite par l'utilisateur est 3.

variables / instructions	nblignes	i	j	
1	4			
2		1		
3			1	
4			écrire	
3			(fin)	
5				retour à la ligne
2		2		
3			1	
4			écrire	
3			2	
4			écrire	
3			(fin)	
5				retour à la ligne
2		3		
3			1	
4			écrire	
3			2	
4			écrire	
3			3	
4			écrire	
3			(fin)	
5				retour à la ligne
2		(fin)		

3 Compléments sur les fonctions

3.1 Appel de fonctions

Les fonctions sont utilisées pour décrire les différentes étapes d'un algorithme complexe. On peut alors *appeler* ces fonctions pour réaliser les étapes de calcul correspondantes : lors de l'appel, on indique grâce aux paramètres les données sur lesquelles la fonction doit travailler, puis on récupère le résultat retourné. Un appel de fonction provoque donc l'exécution complète du corps de cette fonction.

Le passage des paramètres sera décrit ici de manière intuitive. Chaque langage de programmation traite cet aspect d'une manière rigoureuse, ce qui nécessite d'aborder des notions plus détaillées (paramètres formels et effectifs, passage par valeur ou par adresse, portée des paramètres, compatibilité des types, …).

Exemple 1

Décrire un algorithme qui calcule le maximum de 4 réels saisis au clavier. Le calcul du maximum de deux valeurs sera décrit par une fonction.

```
fonction calculerMax2Réels(x : réel, y : réel) : réel
        début
            si x > y                    // f1
                alors
                    res ← x             // f2
                sinon
                    res ← y             // f3
            fsi
            retourne res                // f4
        fin
lexique :
-   x : réel,
-   y : réel,
-   res : réel, maximum trouvé

Algorithme
        début
        maximum ← lire ( )                                  // 1
        pour i de 2 à 4 faire                               // 2
            nombre ← lire ( )                               // 3
            maximum ← calculerMax2Réels(nombre,maximum)     // 4
        fpour
        écrire (maximum)                                    // 5
        fin
lexique
-   maximum : réel, maximum des i premiers nombres réels
-   nombre : réel, ième réel donné
-   i : entier, indice d'itération
```

Schéma de l'évolution de l'état des variables instruction par instruction :
Il est possible d'utiliser les mêmes noms de variables dans différentes fonctions et dans l'algorithme principal. On précise donc à chaque fois à quelle fonction correspond la variable.

On suppose que les valeurs introduites par l'utilisateur sont : 35 80 37 22.

variables instructions	maximum (algo)	nombre (algo)	i (algo)	x (calculer…)	y (calculer…)	x>y (calculer…)	res (calculer…)
1	35						
2			2				
3		80					
4				80	35		
F1						vrai	
F2							80
F4	80						Retourne
2			3				
3		37					
4				37	80		
F1						faux	
F3							80
F4	80						Retourne
2			4				
3		22					
4				22	80		
F1						faux	
F3							80
F4	80						Retourne
2			(fin)				
5	écrire						

Les paramètres fournis lors de l'appel peuvent être des expressions.

Une fonction peut elle-même contenir plusieurs appels de fonctions. Il est également possible d'utiliser le résultat d'une fonction directement dans le paramètre fourni à l'appel d'une autre fonction.

Exemple 2

Un étudiant doit, pour obtenir son diplôme, passer un écrit et un oral dans deux modules. Le coefficient du premier module est le double de celui du second module. La moyenne d'un module, afin de ne pas pénaliser trop les éventuels échecs accidentels, accorde un coefficient double à la meilleure des deux notes obtenues.
On veut décrire un algorithme où, après saisie des quatre notes, la décision finale est affichée (diplôme obtenu si la moyenne est supérieure ou égale à 10, aucun module ne devant avoir une moyenne inférieure à 8).

```
fonction calculerMoyenne(n1 : réel, n2 : réel) : réel
        début
            moy ← (n1+2*n2)/3                   // m1
            retourne moy                        // m2
        fin
lexique :
-   n1 : réel, note de coefficient 1
-   n2 : réel, note de coefficient 2
-   moy : réel, moyenne calculée

fonction calculerNoteModule(n1 : réel, n2 : réel) : réel
        début
            si n1>n2                                    // n1
                alors
                    note ← calculerMoyenne(n2,n1)       // n2
                sinon
                    note ← calculerMoyenne(n1,n2)       // n3
            fsi
            retourne note                               // n4
        fin
```

```
lexique :
-   n1 : réel, première note
-   n2 : réel, deuxième note
-   note : réel, note du module

fonction accorderDiplôme(m1 : réel, m2 : réel) : booléen
        début
            moy ← calculerMoyenne(m2,m1)            // d1
            si moy < 10                             // d2
                alors
                    reçu ← faux                     // d3
                sinon
                    si (m1<8) ou (m2<8)             // d4
                        alors
                            reçu ← faux             // d5
                        sinon
                            reçu ← vrai             // d6
                    fsi
            fsi
            retourne reçu                           // d7
        fin
lexique :
-   m1 : réel, moyenne premier module
-   m2 : réel, moyenne second module
-   moy : réel, moyenne générale
-   reçu : booléen, à vrai si l'étudiant a obtenu son diplôme

Algorithme
    début
        ne_m1 ← lire ( )                                                    // 1
        no_m1 ← lire ( )                                                    // 2
        ne_m2 ← lire ( )                                                    // 3
        no_m2 ← lire ( )                                                    // 4
        obtenu ← accorderDiplôme(calculerNoteModule(ne_m1,no_m1),
                                 calculerNoteModule(ne_m2,no_m2))           // 5
        si obtenu                                                           // 6
            alors écrire(« diplôme obtenu »)                                // 7
            sinon  écrire(« diplôme non accordé »)                          // 8
        fsi
    fin
lexique
-   ne_m1 : réel, note d'écrit du premier module
-   no_m1 : réel, note d'oral du premier module
-   ne_m2 : réel, note d'écrit du second module
-   no_m2 : réel, note d'oral du second module
-   obtenu : booléen, vrai si le diplôme est accordé
```

Schéma de l'évolution de l'état des variables instruction par instruction :

Pour simplifier, on ne détaillera pas les instructions de la fonction « calculerMoyenne », ni les valeurs booléennes des tests.

On suppose que les valeurs introduites par l'utilisateur sont : 12 7 10 8.

Variables / instructions	ne_m1 (algo)	no_m1 (algo)	ne_m2 (algo)	no_m2 (algo)	obtenu (algo)	m1 (Dip...)	m2 (Dip...)	moy (Dip...)	reçu (Dip...)	n1 (Note...)	n2 (Note...)	note (Note...)
1,2,3,4	12	7	10	8								
5										12	7	
n1,n2												10.33
n4						10.33						retourne

5(suite)										10	8	
n1,n2												9.33
n4							9.33					retourne
d1,d2,d4								10				
d6,d7					vrai				vrai			
6,7					écrire							

4 Chaînes de caractères

Le type chaîne permet de décrire des objets formés par la juxtaposition de plusieurs caractères. Dans la plupart des langages de programmation, il existe des « outils » pour les manipuler. Au niveau des algorithmes, nous introduisons quelques fonctions prédéfinies qui correspondent aux « outils » classiquement fournis par les langages. Cette liste partielle peut bien sûr être complétée en fonction des besoins.

4.1 Quelques fonctions sur les chaînes de caractères

Concaténation de chaînes

fonction concat (ch1: chaîne, ch2: chaîne) : chaîne
retourne une chaîne formée par la concaténation de *ch1* et *ch2*. La chaîne résultat est formée de *ch1* suivi de *ch2*.

Longueur d'une chaîne

fonction longueur (ch : chaîne) : entier
retourne la longueur de la chaîne *ch* c'est-à-dire le nombre de caractères dont elle est constituée.

Sous-chaîne

fonction sousChaîne (ch : chaîne, i : entier, l : entier) : chaîne
retourne une sous-chaîne de longueur *l* extraite de la chaîne *ch*, à partir de la position *i*

exemple : sousChaîne ("informatique", 6, 2) retourne la chaîne "ma".

Accès au $i^{ème}$ caractère

fonction ième (ch : chaîne, i : entier) : caractère
retourne le $i^{ème}$ caractère de la chaîne *ch*

Modification du ième caractère

fonction remplace (ch InOut: chaîne, i : entier, c : caractère)
remplace le $i^{ème}$ caractère de la chaîne *ch* par le caractère *c*.

4.2 Exemple

On donne un télégramme mot par mot. On souhaite compter le nombre d'unités de paiement du télégramme sachant qu'il se termine par le mot « stop », qu'un mot de longueur *l* coûte (l ÷10) + 1 unités et que le mot "*stop*" ne coûte rien.

données
la suite des mots composant le télégramme suivi de « stop »

résultats
nombre d'unités de paiement

```
algorithme
        début
        nup ← 0
        mot ← lire ( )
        tant que mot ≠ « stop » faire
                    prixmot ← (longueur (mot)  ÷ 10) + 1
```

```
                nup ← nup + prixmot
                mot ← lire ( )
        ftant
        écrire (nup)
        fin
lexique
-   nup : entier, nombre d'unités de paiement
-   mot : chaîne, ième mot du texte
-   prixmot : réel, prix du ième mot du texte
```

5 Tableaux

5.1 Notion de tableau

Les tableaux servent à désigner une suite finie d'éléments de même type au moyen d'une unique variable. Ces éléments peuvent être des entiers, des réels, des chaînes, … etc. Ils sont stockés dans les différentes cases du tableau, habituellement numérotées de 0 à n-1, si n est la taille du tableau.

Le type d'un tableau précise l'intervalle de définition et le type (commun) des éléments.

tableau type_des_éléments [borne_inférieure .. borne_supérieure]

Dans le module d'algorithmique, nous choisirons toujours la valeur 0 pour la borne inférieure dans le but de faciliter la traduction en C ou en Java. Par exemple, pour un tableau t de 10 entiers , on pourra écrire :

t : tableau entier [0..9]

Un tel tableau peut par exemple contenir les éléments suivants :

0	1	2	3	4	5	6	7	8	9
45	54	1	-56	22	134	49	12	90	-27

Pour accéder à un élément du tableau, il suffit de préciser entre crochets l'indice de la case contenant cet élément. Par exemple, pour accéder au septième élément (49) du tableau d'entiers ci-dessus, on écrit : t[6]. L'instruction suivante affecte à la variable x la valeur du premier élément du tableau, c'est à dire 45 :

x ← t[0]

L'élément désigné du tableau peut être utilisé comme n'importe quelle variable :

t[6] ← 43

Cette instruction a modifié le tableau t :

0	1	2	3	4	5	6	7	8	9
45	54	1	-56	22	134	43	12	90	-27

5.2 Parcours complet d'un tableau

La plupart des algorithmes basés sur les tableaux utilisent des itérations permettant de faire un parcours complet ou partiel des différents éléments du tableau. De tels algorithmes établissent le résultat recherché par récurrence en fonction des éléments successivement rencontrés.

Les répétitions inconditionnelles sont le moyen le plus simple de parcourir complètement un tableau.

Exemple 1

Dans l'exemple suivant, la fonction affiche un à un tous les éléments d'un tableau de n éléments :

```
fonction écrireTableau(n : entier, tab : tableau entier [0..n-1])
        début
            pour i de 0 à n-1 faire              // f1
                écrire(tab[i])                   // f2
            fpour
        fin
lexique
-   i : entier,  indice d'itération
-   n : entier, taille du tableau
-   tab : tableau entier [0..n-1]

Algorithme
    début
        n ← lire ( )                             // 1
        tab ← lire ( )                           // 2
        écrireTableau(n, tab)                    // 3
    fin
lexique
-   n : entier, taille du tableau
-   tab : tableau entier [0..n-1]
```

Schéma de l'évolution de l'état des variables instruction par instruction :

On suppose que le tableau saisi, de taille 3, contient : 7 10 8.

variables / instructions	N	tab			n	i	tab		
		tab[0]	tab[1]	tab[2]			tab[0]	tab[1]	tab[2]
1	3								
2		7	10	8					
3					3		7	10	8
f1						0			
f2							écrire		
f1						1			
f2								écrire	
f1						2			
f2									écrire
f1						(fin)			

Exemple 2

La fonction suivante multiplie par 2 tous les éléments d'un tableau.

```
fonction doublerTableau(n : entier, t InOut : tableau entier [0..n-1])
        début
            pour i de 0 à n-1 faire              // 1
                t[i] ← t[i]*2                    // 2
            fpour
        fin
lexique :
-   n : entier, taille du tableau
-   t : tableau entier [0..n-1], tableau modifiable
```

Schéma de l'évolution de l'état des variables instruction par instruction :

On suppose que l'appel de fonction est « doublerTableau(3, tab) », où tab est un tableau de taille 3 qui contient : 7 10 8.

variables / instructions	n	tab			n	i	t		
		tab[0]	tab[1]	tab[2]					
…	3	7	10	8					
appel		←	←	←	3				
1						0			
2		14							
1						1			
2			20						
1						2			
2				16					
1						(fin)			
…									

5.3 Parcours partiel d'un tableau

Certains algorithmes sur les tableaux se contentent de parcourir successivement les différents éléments du tableau jusqu'à rencontrer un élément satisfaisant une certaine condition. Un tel parcours partiel est le plus souvent basé sur une répétition conditionnelle.

Exemple

On cherche ici à savoir si un tableau saisi au clavier n'est constitué que d'entiers positifs :

```
algorithme
    début
        tab ← lire()
        i ← 0
        positif ← vrai
        tant que positif et i < n faire
            si tab[i] < 0
                alors positif ← faux
            fsi
            i ← i + 1
        ftant
        si positif
            alors écrire("tableau d'entiers naturels")
            sinon écrire("tableau d'entiers relatifs")
        fsi
    fin
```

lexique

- i : entier, indice d'itération
- n : entier, taille du tableau
- tab : tableau entier [0..n-1]
- positif : booléen, vrai si aucun entier négatif n'a été détecté

5.4 Parcours imbriqués

Certains algorithmes sur les tableaux font appel à des boucles imbriquées : la boucle principale sert généralement à parcourir les cases une par une, tandis que le traitement de chaque case dépend du parcours simple d'une partie du tableau (par exemple toutes les cases restantes), ce qui correspond à la boucle interne.

Exemple

La fonction suivante calcule, pour chaque case d'un tableau, le nombre de cases suivantes qui contiennent un élément strictement supérieur. Les résultats sont placés dans un tableau.

```
fonction nbSuccesseursSup(n : entier, t : tableau entier [0..n-1]) : tableau entier [0..n-1]
    début
        pour i de 0 à n-1 faire
            res[i] ← 0
        fpour
        pour i de 0 à n-2 faire
            pour j de i+1 à n-1 faire
                si t[i]<t[j]
                    alors
                        res[i] ← res[i]+1
                fsi
            fpour
        fpour
        retourne res
    fin
```

lexique :

- n : entier, taille du tableau
- t : tableau entier [0..n-1],
- res : tableau entier [0..n-1], tableau résultat (case i contient le nombre de cases de t d'indice supérieur à i qui contiennent un élément supérieur à t[i])
- i : entier, indice de la boucle principale (parcours pour remplir res)
- j : entier, indice de la boucle interne (parcours des cases restantes de t)

5.5 Tableaux multidimensionnels

Les cases d'un tableau à une dimension sont indicées de manière consécutive (cases « alignées »). Il est possible de disposer ces cases selon des grilles (tableaux à deux dimensions), des cubes (tableaux à trois dimensions), …

Les algorithmes les plus simples sur ces tableaux utilisent néanmoins en général des boucles imbriquées : chaque niveau de boucle correspond au parcours selon une dimension.

Le type d'un tableau précise l'intervalle de définition selon chaque dimension.

tableau type_des_éléments [borne_inf_dim1 .. borne_sup_dim1, borne_inf_dim2 .. borne_sup_dim2, …]

Tableaux à deux dimensions

Ces tableaux sont faciles à se représenter comme une grille ayant un certain nombre de lignes (première dimension) et un certain nombre de colonnes (seconde dimension).

tableau type_des_éléments [0 .. nb_lignes-1, 0 .. nb_colonnes-1]

Un tel tableau, avec 5 colonnes et 3 lignes, peut par exemple contenir les éléments suivants :

	0	1	2	3	4
0	45	54	1	-56	22
1	64	8	54	34	2
2	56	23	-47	0	12

Pour accéder à un élément du tableau, il suffit de préciser entre crochets l'indice de la case contenant cet élément, et ce pour chacune des dimensions. Par exemple, pour accéder à l'élément 23 du tableau d'entiers ci-dessus, on écrit : t[2,1]. L'instruction suivante affecte à la variable x la valeur du premier élément du tableau, c'est à dire 45.

x ← t[0,0]

L'élément désigné du tableau peut alors être utilisé comme n'importe quelle variable :

t[2,1] ← 43

Cette instruction a modifié le tableau :

45	54	1	-56	22
64	8	54	34	2
56	43	-47	0	12

Exemple : calcul de la somme

```
fonction somme(li : entier, co : entier, t : tableau entier [0..li-1,0..co-1]) : entier
    début
        s ← 0
        pour i de 0 à li-1 faire
            pour j de 0 à co-1 faire
                s ← s + t[i,j]
            fpour
        fpour
        retourne s
    fin
```

lexique :

- li : entier, nombre de lignes du tableau
- co : entier, nombre de colonnes du tableau
- t : tableau entier [0..li-1,0..co-1], tableau dont on cherche l'élément maximal
- s : entier, somme des éléments déjà parcourus
- i : entier, indice d'itération sur les lignes
- j : entier, indice d'itération sur les colonnes

5.6 Recherche dichotomique

La fonction *rechercheDicho* recherche un élément dans un tableau trié et retourne l'indice d'une occurrence de cet élément (ou –1 en cas d'échec). Une telle recherche peut être réalisée de manière séquentielle ou dichotomique. Nous développons ici la version dichotomique qui est la plus efficace en temps d'exécution.

On compare l'élément cherché à celui qui se trouve au milieu du tableau. Si l'élément cherché est plus petit, on continue la recherche dans la première moitié du tableau sinon dans la seconde. On recommence ce processus sur la moitié. On s'arrête lorsqu'on a trouvé ou lorsque l'intervalle de recherche est nul.

Exemple :

Recherchons l'entier 46 puis 10 dans le tableau d'entiers suivant défini sur l'intervalle [3. .10] :

5	13	18	23	46	53	89	97
3	4	5	6	7	8	9	10

- recherche de l'entier 46 :
  étape 1 : comparaison de 46 avec t [6], t [6] < 46 => recherche dans [7..10]
  étape 2 : comparaison de 46 avec t [8], t [8] > 46 => recherche dans [7 .. 7]
  étape 3 : comparaison de 46 avec t [7], t [7] = 46 => l'élément cherché se trouve à l'indice 7
  Remarque : le déroulement serait très différent si c'était t[6] qui valait 46.

- recherche de l'entier 10 :
  étape 1 : comparaison de 10 avec t [6], t [6] > 10 => recherche dans [3..5]
  étape 2 : comparaison de 10 avec t [4], t [4] > 10 => recherche dans [3..3]
  étape 3 : comparaison de 10 avec t [3], t [3] < 10 => recherche dans [4..3]
  Attention : borne inférieure > borne supérieure
  => arrêt, on a pas trouvé l'élément cherché.

```
fonction rechercheDicho(e : entier, n : entier, t : tableau entier [0..n-1]) : entier
    début
        trouve ← faux
        debut ← 0
        fin ← n-1
        tant que debut≤fin et non trouve faire
            i ← (debut+fin)÷2
            si t[i] = e
                alors trouve ← vrai
                sinon
                    si t[i] > e
                        alors fin ← i-1
                        sinon debut ← i+1
                    fsi
            fsi
        ftant
        si trouve
            alors retourne i
            sinon retourne –1
        fsi
    fin
```

lexique :

- n : entier, taille du tableau
- t : tableau entier [0..n-1], tableau trié par ordre croissant
- e : entier, élément recherché
- trouve : booléen, faux tant que l'élément cherché n'est pas trouvé
- i : entier, indice de la case observée (case du milieu)
- debut: entier, indice minimum de la zone de recherche
- fin: entier, indice maximum de la zone de recherche

6 Variables composites

On appelle *type composite* un type qui sert à regrouper les caractéristiques (éventuellement de types différents) d'une valeur complexe. Chaque constituant est appelé *champ* et est désigné par un *identificateur de champ*. Le type composite est parfois aussi appelé *structure* ou *produit cartésien*. On utilise chaque identificateur de champ comme sélecteur du champ correspondant.

6.1 Présentation du type composite

Définition lexicale

c : <chp1 : type1, ... , chpn : type n>

où *c* est une variable composite dont le type est décrit,
chp1, *chp2*, ..., *chpn* sont les noms des champs de la variable composite *c*,
type1 est le type du champ *chp1*,..., *typen* le type du champ *chpn*.

Exemples:
- date: <jour: entier, mois: chaîne, année: entier >
- personne: <nom: chaîne, prénom: chaîne, datenaiss: <jour: entier, mois: chaîne, année: entier>, lieunaiss : chaîne>

Sélection de champ

sélection du champ chpi de c : *c.chpi*

c.chpi est une variable de type *typei* et peut être utilisé comme toute variable de ce type. On utilise chaque identificateur de champ comme sélecteur du champ correspondant.

Exemples:
date désigne une variable de type composite <jour: entier, mois: chaîne, année: entier >.
date.jour désigne une variable de type entier .
personne.datenaiss.année est aussi une variable de type entier.

Construction d'une variable composite par la liste des valeurs des champs :

c ← (c1, c2, ..., cn) où *c1* : *type1* , *c2* : *type2*, ..., *cn* : *typen*.

La liste des valeurs des champs est donnée dans l'ordre de la description du type composite.

Exemples:
Soit les deux types composites :
Date = <jour : entier, mois: chaîne, année: entier>[1]
Personne = <nom: chaîne, prénom: chaîne, datenaiss: Date, lieunaiss: chaîne>
et les variables :
père, fils : Personne ,
date : Date .
On peut écrire :
date ← (1, "janvier", 1995)
fils ← (père.nom, "Paul", (14, "mars", 1975), "Lyon")

[1] On peut nommer un type la première fois qu'il apparaît et ensuite on utilise ce nom pour les autres variables de même type.

Modification de la valeur d'un champ :

c.chpi ← ci où *ci* est de type *typei*, la valeur *ci* est affectée au champ *chpi* de *c*.

Exemple:

date.jour ← 30

Autres "opérations" :

c ← lire( )

écrire(c)

c1 ← c2 où c1 et c2 sont deux variables composites du même type.

6.2 Exemple

Dans le service qui affecte les nouveaux numéros INSEE, on veut écrire l'algorithme de la fonction qui, à partir des renseignements nécessaires sur une personne donnée et du dernier numéro personnel affecté par ce service, crée un nouveau numéro INSEE.

Les renseignements sur la personne sont fournis sous la forme d'une variable composite formée de:

- nom
- prénom
- date de naissance sous la forme d'un triplet (jour, mois, année)
- code ville
- département
- sexe : 'M' pour masculin, 'F' pour féminin

Le numéro d'INSEE sera de type composite, formé de :

- code: 1 pour masculin , 2 pour féminin
- année de naissance (seulement les 2 derniers chiffres)
- mois de naissance sous la forme d'un entier
- numéro du département de naissance
- code de la ville de naissance
- numéro personnel

Le numéro personnel est obtenu en ajoutant 1 au dernier numéro personnel affecté par le service.

```
algorithme
fonction inseeCreer (personne : EtatCivil, dernier_numéro : entier) :Insee
        début
        si personne.sexe = 'M'
            alors  code ←1
            sinon  code ← 2
        fsi
        numéro_insee ←  (code, personne. datenaiss.année mod 100, personne.datenaiss.mois ,
                              personne.depnaiss, personne.villenaiss , dernier_numéro + 1)
        retourne numéro_insee
        fin
```

lexique

- Etatcivil = <nom: chaîne, prénom: chaîne, datenaiss: Date, depnaiss : entier, villenaiss : entier, sexe : caractère >
- personne : Etatcivil, personne qui souhaite un numéro INSEE
- dernier_numéro : entier, dernier numéro personnel affecté
- code : entier, code pour représenter le sexe
- Insee = < codesexe : entier, annaiss : entier, moisnaiss : entier, depnaiss : entier, villenaiss : entier, numéro : entier>
- numéro_insee : Insee, numéro INSEE affecté.