Ines A.

# ALGORITHMIQUE 1

## CHAPITRE 1 : introduction

/

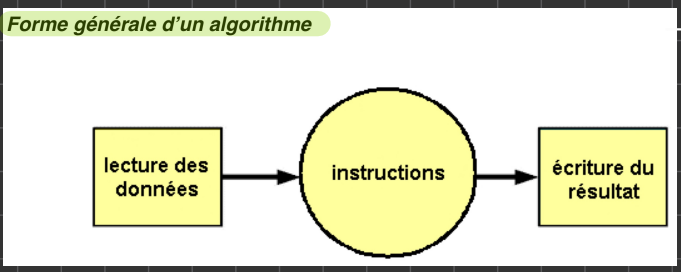
## CHAPITRE 2 : algorithmes séquentiels

- def **algorithme séquentiel/linéaire** => suite d'instructions qui s'effectuent entièrement dans leur ordre d'écriture et une seule fois
  => contient pas d'alternatives ou d'iterations

**données** => y sont associés des opérations/actions
**variables** => entités permettant de stocker la valeur des données dans mémoire de l'ordi
**types** => genre de données pouvant ê stocké par la variable

↳ **type entier** ensemble $\mathbb{Z}$
**type réel (flottants)** ensemble $\mathbb{R}$
**type booléen** T/F
**type chaîne (de caract.)** suite de caract. alpha-numériq. "..."

- *Forme générale d'un algorithme*




→ selon le modèle suivant :

```
algorithme [nom]
        // déclaration des données
        // lecture des données
        // instructions agissant sur les données
        // écriture du résultat
fin
```

↳ **déclaration des données** : section donnant la liste des variables utilisées (en série de lignes) nom de variables : type
(pas d'espaces)

**lecture des données** : séquence d'instruction par laquelle l'ordi prend connaissance des données et demande à l'utilisateur de l'algo de fournir les valeurs des variables (affecta° externe) lire variable

**instructions agissant sur les données** : contient des instructions qui ont pour but de transformer les données, de calculer des nouvelles valeurs et de les stocker, s'il faut, dans de nouvelles variables (affecta° internes) variables ← expression

Les principaux opérateurs arithmétiques sont les suivants :

| | |
|---|---|
| + | addition |
| – | soustraction |
| * | multiplication |
| / | division réelle |
| DIV | division entière |
| MOD | reste de la division entière |

Si leur usage s'avère nécessaire, les fonctions mathématiques peuvent être écrites sous la forme habituelle : sin(x), cos(x), log(x), exp(x) pour l'exponentielle de x, et sqrt(x) pour la racine carrée. Le calcul d'exposant – assez rare dans les programmes informatiques – peut se noter sous la forme x^y (x élevé à la puissance y).

écriture du résultat : communique le(s) résultat(s) attendu(s) par l'utilisateur
écrire variable

commentaire : // ceci est un commentaire
# "
/* " */

- tracer un algo => suivre à la trace l'évolution de toutes les variables d'un algo.
=> utilité : pouvoir valider un code + comprendre et repérer une erreur de logique

```
1   algorithme somme
2       x, y, somme : entier
3       lire x, y
4       somme ← x + y
5       écrire somme
6   fin
```

EXEMPLE

Exemple pour l'algorithme somme ci-dessus, en supposant que les données entrées sont 18 et 24 :

| ligne | x | y | somme |
|---|---|---|---|
| 2 | // pas de contenu | // pas de contenu | // pas de contenu |
| 3 | 18 | 24 | // pas de contenu |
| 4 | 18 | 24 | 42 |

- mots reservés => mots, comme algorithme, lire, écrire et fin, faisant partis de structures de contrôle et sont déconseillés à utiliser pour autre chose.

- qualité d'un algorithme => 1) la validité : doit fournir le résultat attendu
2) la lisibilité : doit être compréhensible à sa simple lecture, clair et lisible!
3) la performance : doit être efficient / rapide pour la résolution d'un prolem donné.

## CHAPITRE 3 : alternatives

- la structure alternative de base : si - sinon - fin si

```
si condition alors → ignoré si la condition est F
    // bloc d'instructions qui sont exécutées lorsque la condition est vraie
sinon → ignoré si la condition est V
    // bloc d'instructions qui sont exécutées lorsque la condition est fausse
fin si
```

↳ la condition => expression à valeur booléenne
ou " calculée à pd de l'algo et qui aura une valeur finale V/F

- alternative "sans alternative" => éxecution d'instruction soumise à une condition mais rien n'est prévu/précisé si la condition n'est pas satisfaite

↳ structure :

```
si condition alors
    // bloc d'instructions qui sont exécutées lorsque la condition est vraie
fin si
```

càd le sinon est facultatif

- alternatives multiples => plusieurs blocs d'instructions dont un seul sera exécuté

↳ structure :

```
si condition1 alors
        // bloc d'instructions qui sont exécutées lorsque condition1 est vraie
sinon si condition2 alors
        // bloc d'instructions qui sont exécutées lorsque condition1 est
        // fausse et condition2 est vraie
sinon si condition3 alors
        // bloc d'instructions qui sont exécutées lorsque condition1 et
        // condition2 sont fausses et la condition3 est vraie
        ...
sinon ——> peut ê facultatif
        // bloc d'instructions qui ne sont exécutées que lorsque toutes
        // les conditions précédentes sont fausses
fin si
```

→ les conditions sont évaluées en cascade et la 1ère étant V sera exécutée.

→ si aucune des conditions est V, alors le sinon va ê exécuté

- structure du "selon que" ⇒ cas où nbre d'alternatives est très élevé

```
selon que variable vaut
        valeur(s)1 : instruction(s)1
        valeur(s)2 : instruction(s)2
        valeur(s)3 : instruction(s)3
        ...
        autre : instruction(s) ——> aussi facultatif
fin selon
```

une m valeur ne peut pas ê dans plusieurs listes

→ le bloc d'instruction exécuté est celui correspondant à la 1e liste de valeurs contenant la valeur de la variable

→ si aucun bloc, alors autre sera exécuté

- variables booléennes ⇒ 2 possibilités de contenus : V ou F
  ⇒ pas leur donner un contenu par une affectation externe (lire), plutôt par affectation interne via une expression qui a elle-m une valeur booléenne

- opérateurs booléens ⇒ agit sur une/plusieurs expressions booléennes pour construire une expression + complexe ayant également une valeur booléenne
  ⇒ les 3 principaux : ET (conjonction)
  OU (disjonction inclusive)
  NON (négation)
  } voir tables de vérité
  (utilisation des parenthèses si nécessaire en fct des priorités)

- évaluation court-circuitée

## CHAPITRE 4 : modularisation

- algorithme avec paramètres ⇒ 1) remplacer les instructions de lecture des données par des paramètres en entrée (la 1ère étape)
  ↳ variables dont le contenu est communiqué lors de l'appel de l'algo.
  2) dès que variables affectées → l'algo peut s'enclencher sans nécessiter de lecture de données

**Exemple**

Nous allons adapter l'algorithme *maximum* écrit précédemment en remplaçant la lecture des nombres par des paramètres en entrée, ce qui donne ceci :

déclaration de données

```
1   algorithme maximum(x, y : entier)
2       max : entier
3       si x > y alors
4           max ← x
5       sinon
6           max ← y
7       fin si
8       écrire "Le maximum vaut", max
9   fin
```

l'algo fctionne que si x et y ont reçu une valeur ; ce fait lors de l'appel de l'algo.

- appel d'un algo. d'un autre algo → suffit d'écrire son nom accompagné de valeurs/variables en ⩵ nbre et type que ceux figurant dans son entête

EXEMPLE

```
1  algorithme exemple
2      a, b : entier
3      lire a, b
4      maximum(a, b)
5      maximum(10, b)
6      maximum(10, 20)
7  fin
```

4 : appel valide si les 2 expressions entre (...) ont reçu une valeur ayant le type attendu
4 : provoque de façon implicite les 2 affectations internes : x ← exp. 1 et y ← exp. 2

- valeur de retour ⇒ éliminer à présent l'écriture au niv. de l'algo. appelé (la 2e étape)
  ⇒ récupérer le résultat d'un algo. sans qu'il s'affiche → utilisation de la valeur de retour
  ⇒ retourner ... (mot réservé)

EXEMPLE

```
1  algorithme maximum(x, y : entier) → entier
2      si x > y alors
3          max ← x
4      sinon
5          max ← y
6      fin si
7      retourner max
8  fin
```

1 : → signifie qu'à la fin de l'algo., une valeur de ce type va être affichée
7 : renvoie la valeur au niv. de l'appel de l'algo.

↳ l'appel d'un algo. avec une valeur retour diffère !

EXEMPLE

```
1  algorithme exemple
2      a, b : entier
3      lire a, b
4      écrire maximum(a, b)
5      écrire maximum(10, b)
6      écrire maximum(10, 20)
7  fin
```

4-6 : utilisation du nom de l'algo. comme une exp. et plus comme une instruction indé.

→ remarques 1) l'instruction retour en fin d'algo. car les lignes qui suivaient seraient ignorées (du "code mort")

2) possible d'avoir qu'une seule instruction retourner (les autres après la 1ère seront ignorées)

3) retourner ... → soit variable contenant valeur de retour
   soit une exp.

4) variables utilisées au niv. d'un algo. sont locales càd qu'elles sont inconnues du corps de l'algo. où elles apparaissent
   + ça permet de pouvoir utiliser les ⩵ noms de variables dans ≠ algo.

5) avantage de couper un algo. en sous-algo → déléguer les tâches et les séparer de façon claire en fct de leur spécificité

Ines A.

# CHAPITRE 5: variables structurées

- **variable structurée** ⇒ permet de pouvoir manier l'ensemble de données éparpillées en les regroupant dans une seule donnée unifié

↳ "avant" ; utilisation de variables de types "simples" ex entier, réel, chaine de caract., booléen qui ne peuvent contenir qu'une seule valeur à la fois

MAIS certains problms, les données sont constituées de plusieurs parties ex adresse, date ...

- **type structuré** ; pour pouvoir utiliser une variable structurée :

↳ 1) définir son type càd lisk énonçant les ≠ champs de la variable et les types de ces ≠ champs

ex

```
type date
        jour : entier
        mois : entier
        année : entier
fin type

type moment
        heure : entier
        minute : entier
        seconde : entier
fin type
```

```
type adresse
        rue : chaîne
        numéroMaison : chaîne
        codePostal : entier
        ville : chaîne
        pays : chaîne
fin type
```

↳ dés que ces types ont été définis, ils peuvent ê utilisés comme des types "simples".

↳ **champ d'une variable structurée** ⇒ peut lui-m ê structuré

ex

```
type carteIdentité
        nom : chaîne
        prénom : chaîne
        dateNaissance : date
        domicile : adresse
        numRegistre : chaîne
fin type
```

- **déclaration d'une variable structurée** :

```
nom de variable : type
```

- **variable structurée** ⇒ collection de variables simples

↳ accéder à un champs particulier d'une variable structurée ⇒ utilisation de la **notation pointée**

ex

```
dateAnniversaire.jour ← 5
heureRéveil.heure ← 7
adresseEcole.rue ← "Rue Royale"
écrire carteIdentité.nom
retourner carteIdentité.domicile.codePostal
```

↳ on peut manier l'ensemble des données d'une variable structurée ;

ex

```
dateDuJour ← dateAnniversaire
retourner heureRéveil
maNouvelleAdresse ← monAdresse
carteIdentité.domicile ← maNouvelleAdresse
```

↳ son retour de valeur est très utile car permet de retourner un ensemble de valeurs en une seule fois.

Ines A.

# CHAPITRE 6 : itérations

- **boucles "pour"** => utilisées lorsque le nbre précis de répétitions d'une série d'instructions est connu à l'avance
  => forme générale :

```
pour var de a à b faire
    [ instructions]
fin pour
```

var = variable
a = valeur de départ } doivent être connus avant de lancer la boucle
b = " d'arrivée

=> signifie que ∀ les valeurs ℤ de la variable comprises entre a et b, les instructions à l'intérieur de la boucle vont être exécutées.

↳ var = **variable de contrôle** de la boucle => pas besoin d'être connue avant de lancer
=> implicitmnt affectée à a au début de la boucle
=> peut apparaître dans les instructions (boucle ne produit pas tjs le m résultat car dépendra de var)
=> sa valeur finale est b en fin de boucle ; peut être ≠ selon langage de progra. -> faut donc considérer sa valeur comme IND.

(plein d'ex dans le xylla.)

- **boucles "pour" avec pas** => sans mention d'un "pas", variable de contrôle est automtqmnt incrémentée d'une unité.
  => si on veut lui incrémenter autre chose :

```
pour var de a à b par pas faire
    [ instructions]
fin pour
```

=> signifie que var va être à chaque itération incrémentée de la valeur contenue dans la variable pas (celle-ci peut être ⊖)

- **les bonnes pratiques** => ne jamais modifier la variable de contrôle par des instructions à l'intérieur de la boucle
  => faire un choix cohérent des variables a, b et pas ; si a<b -> pas doit être ⊕
  si a>b -> pas doit être ⊖
  ↳ hors cette cohérence, la boucle n'est pas exécutée.

- **boucles "tant que"** => utilisées lorsque nbre d'itérations n'est pas forcément connu à l'avance.
  => l'arrêt de la boucle dépend d'une condition qui doit changer de valeur booléenne
  => forme générale :

```
tant que condition faire
    [ instructions]
fin tant
```

condition = expression booléenne contenant une variable présente dans les instructions sinon ne changera jms de valeur → boucle infinie

=> signifie que tant que la condition énoncée dans l'en-tête est V, on va répéter les instructions à l'intérieur de la boucle, s'arrête quand condition F.

↳ la condition est généralmnt V, sinon on ne rentrerait pas dans la boucle (pas exécutée)

(peut remplacer une boucle "pour" par une boucle "tant que", le contraire est - évident)

↳ ≠ fondamentales ! -> la variable doit être initialisée avant la boucle + son incrémentation doit s'écrire explictmnt dans le code.

## ↳ qq algorithmes modèles utilisant la boucle "tant que" :

### 1) lecture d'une série de valeurs avec valeur sentinelle => permet de lire une série de données dont la fin est signalée par une valeur sentinelle (=> valeur spéciale qui ne fait pas logiquement partie de l'ensemble des autres valeurs, à choisir au cas par cas !)

ex

Prenons pour exemple une série de cotes d'interrogation, dont les valeurs sont des entiers compris entre 0 et 20. On peut choisir comme valeur sentinelle –1, qui n'est pas une possibilité de cote. Voici un exemple d'une telle série :

12 15 10 8 7 9 15 10 15 17 12 1 14 10 6 5 18 20 15 10 –1

Pour exemple, voici un algorithme qui compte le nombre de cotes encodées :

```
1   algorithme nombreCotes
2       cote, cpt : entier
3       cpt ← 0
4       lire cote
5       tant que cote ≠ –1 faire
6           cpt ← cpt + 1
7           lire cote
8       fin tant
9       écrire « Le nombre de cotes est », cpt
10  fin
```

Quelques points importants :

- ligne 4 : la première lecture se fait hors boucle ; c'est nécessaire pour couvrir le cas particulier de la série « vide » qui ne contiendrait alors que la valeur sentinelle
- ligne 7 : la lecture de la valeur suivante se fait juste avant la fin de la boucle ; lorsque la valeur sentinelle –1 est atteinte, la condition de la ligne 5 devient fausse et la boucle se termine

### 2) parcours des chiffres d'un nbre

ex

Voici un exemple type d'algorithme parcourant un nombre donné, les chiffres étant parcourus de droite à gauche ; cet algorithme compte le nombre de 5 contenus dans le nombre lu.

```
1   algorithme nombreDeCinq
2       nombre, cpt, chiffre : entier
3       cpt ← 0
4       lire nombre
5       tant que nombre ≠ 0 faire
6           chiffre ← nombre MOD 10
7           si chiffre = 5 alors
8               cpt ← cpt + 1
9           fin si
10          nombre ← nombre DIV 10
11      fin tant
12      écrire « Le nombre de 5 est », cpt
13  fin
```

Voici le traçage de cet algorithme si le nombre lu est 57556 :

| ligne | nombre | chiffre | cpt |
|---|---|---|---|
| 2 | // pas de contenu | // pas de contenu | // pas de contenu |
| 3 | // pas de contenu | // pas de contenu | 0 |
| 4 | 57556 | // pas de contenu | 0 |
| 6 | 57556 | 6 | 0 |
| 10 | 5755 | 6 | 0 |
| 6 | 5755 | 5 | 0 |
| 8 | 5755 | 5 | 1 |
| 10 | 575 | 5 | 1 |
| 6 | 575 | 5 | 1 |
| 8 | 575 | 5 | 2 |
| 10 | 57 | 5 | 2 |
| 6 | 57 | 7 | 2 |
| 10 | 5 | 7 | 2 |
| 6 | 5 | 5 | 2 |
| 8 | 5 | 5 | 3 |
| 10 | 0 | 5 | 3 |

L'algorithme se termine par le message « Le nombre de 5 est 3 ».

remarque 1) rappel que les nbres sont stockés en binaire dans la mémoire de l'ordi Donc le parcours des chiffres d'un nbre paraît évident avec son écriture en base 10 mais devient - banal en binaire !

# CHAP 7 : tableaux

- tableau (en info) => structure permettant de stocker une série de variables de m type sous une seule appelation, l'emplacemnt des variables étant différencié et déterminé par un indice

- taille d'un tableau => nbre de variables qu'il peut contenir
  longueur

↳ décalage entre l'indice d'un élément et son numéro d'ordre.

↳ l'indice du 1er élément = 0 (comme en Java et Python où tableaux ≈ listes)
l'indice du 2e " = 1
⋮
l'indice du dernier élément = -1

↳ accéder/utiliser un élément d'un tableau : tab[i] – désigne l'élément d'indice i du tableau tab.

• déclaration d'un tableau ⇒ mentionner son nbre d'éléments et le type de ceux-ci.

ex

```
1   tab : tableau de 10 entiers → crée un tableau de 10 entiers indicé de 0 à 9
2   prix : tableau de 100 réels →  "  "  "  "  100 réels  "  de 0 à 99
3   calendrier : tableau de 365 date(s) → ...
```

→ le contenu d'un tableau est indéfini lors de sa déclaration (doit être explicitmnt initialisé avant de pouvoir être utilisé)

↳ dans la plupart des cas, on va écrire des algo pour lesquels un tableau est reçu en paramètre et dont le contenu a déjà été précisé.

ex algo montrant comment créer et initialiser un tableau avec des valeurs entrées par utilisateur :

```
1   algorithme initialisationTableau
2       i : entier
3       tab : tableau de 10 entiers
4       pour i de 0 à 9 faire
5           écrire « entrer l'élément d'indice », i
6           lire tab[ i ]
7       fin pour
8   fin
```

→ en pratique, on n'initialise jamais un tableau manuelmnt mais plutôt par des procédures automatq. ou par lecture de données dans fichiers

ex initialiser tableau de taille 1000 avec des 0 :

```
1   algorithme initialisationTableauAvecZéro
2       i : entier
3       tab : tableau de 1000 entiers
4       pour i de 0 à 999 faire
5           tab[ i ] ← 0
7       fin pour
8   fin
```

• parcours d'un tableau ⇒ lorsqu'un tableau est déclaré, sa taille est fixée et ne peut pas être modifié au cours d'un algo.
⇒ tab.taille – connaître la taille d'un tableau déclaré

↳ notation pointée similaire à celle utilisée pour variables structurées et est utilisée dans programmation orientée objet.

↳ considère que taille = attribut/propriété d'un tableau qui peut être récup par cette expression

ex tableau en paramètre, supposé déjà initialisé, et affiche son contenu à l'écran :

```
1   algorithme affichageTableau(tab : tableau d'entiers)
2       i : entier
3       pour i de 0 à tab.taille – 1 faire
4           écrire tab[ i ]
5       fin pour
6   fin
```

→ façon de faire plutôt commode ; cet algo peut fonctionner pour n'importe quel tableau d'entier indépamnt de la taille.

• taille physique et taille logique ⇒ un tableau n'est pas facilmnt utilisé dans sa totalité ;
1) nbre de cases du tableau utilisé = taille logique
2) taille totale = taille physique

ex

Voici par exemple un tableau dont seules les cases d'indices 0 à 6 ont été affectées.
Sa taille physique est 10, mais sa taille logique est 7.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 50 | 45 | 15 | 35 | 20 | 80 | 25 | ? | ? | ? |