Secteur Tertiaire Informatique

Filière « Etude et développement »

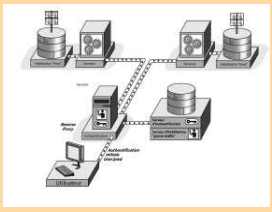
Algorithmie

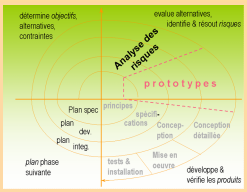
Notions d'algorithmie

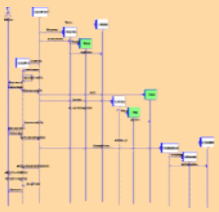
Apprentissage

Mise en pratique

Evaluation







Version	Date	Auteur(s)	Action(s)
1.0	29/10/18	Ludovic Domenici	Création du document
1.1	04/12/19	Alexis Rouyer	Ajout du préambule

Objectifs

À l'issue de ce cours les apprenants sont capables de créer des algorithmes sur des problématiques basiques.

Pré requis

Aucun.

Outils de développement

Méthodologie

Ce document est utilisé par le formateur lors d'un cours en présentiel afin d'élaguer et donner une visualisation globale de l'algorithmie.

Mode d'emploi

Symboles utilisés :

 Renvoie à des supports de cours, des livres ou à la documentation en ligne constructeur.

 Propose des exercices ou des mises en situation pratiques.

 Point important qui mérite d'être souligné !

Ressources

Lectures conseillées

Support de cours, apprentissage et exercices.

1. PREAMBULE

Avant de commencer à aborder les notions d'algorithmique, il convient de préciser qu'il s'agit avant tout de trouver une logique propre à soi. Il faut s'approprier des notions et des processus de langage permettant de transcrire simplement des problèmes complexes. Chaque problème se décompose en plusieurs étapes. Avec l'expérience et la pratique, les étapes simples pourront être raccourcies, donnant à vos algorithmes plus d'efficacité.

Dans un souci de clarté et d'apprentissage, il convient donc dans un premier temps de tout décomposer, en actions basiques et simples puis de monter en puissance. L'algorithmique doit devenir une habitude, un mode de pensée, une manière de s'exprimer.

Au final, faire des algorithmes est synonyme de réfléchir et raisonner. A la différence près qu'il s'agira de mettre sur support vos pensées et raisonnements.

Voici quelques exemples d'entraînement à la réflexion à effectuer avant de continuer la suite.

Cas n°1
Entrée : J'ai les mains sales, je suis devant la salle de bain
Sortie : Je sors de la salle de bain, les mains propres

Cas n°2
Entrée : Je dois sortir la voiture du garage.
Sortie : La voiture est sortie du garage.

Cas n°3
Entrée : J'ai faim. Je mangerai bien une soupe s'il en reste.
Sortie : Il n'y avait plus de soupe, j'ai mangé des pâtes.

Cas n°4
Entrée : Je dois régler mes courses avec ma carte bleue
Sortie : Après m'être trompé une fois dans le code, j'ai réussi à payer

Cas n°5
Entrée : Je dois vérifier si le mot de passe rentré par un utilisateur est conforme à ce que j'attends
Sortie : J'indique à l'utilisateur si son mot de passe est correct, il doit recommencer en cas d'erreur

Cas n°6
Entrée : Je veux jouer une musique de Noël tant qu'il y a une personne dans l'allée de ma maison
Sortie : La musique est jouée quand quelqu'un est présent

Cas n°7
Entrée : Je dois ranger les chocolats du calendrier de l'avent pour mes enfants
Sortie : J'ai rempli les 24 cases de la boite

Cas n°8
Entrée : Je suis serveur. Un client vient d'arriver.
Sortie : J'ai pris sa commande que je transmets en cuisine

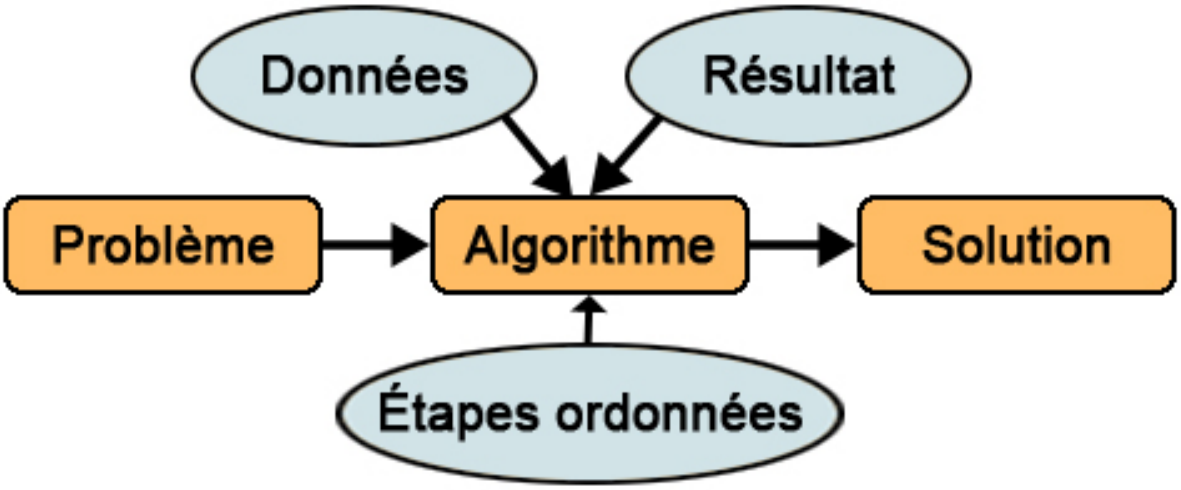
Cas n°9
Entrée : Les plombs ont sauté. Je n'ai pas de lumière, le disjoncteur est à la cave
Sortie : La lumière est revenue

Cas n°10
Entrée : J'ai un CV à rédiger
Sortie : Mon CV est rédigé

2. INTRODUCTION

Un algorithme est un méta-langage de programmation qui sous forme de langage humain va permettre de décomposer en différentes instructions simples, claires et élémentaires une problématique afin d'obtenir un résultat.

Grossièrement c'est une démarche à suivre pour obtenir un résultat.



Par exemple pour définir l'indice de masse corporelle d'une personne le calcul ce fait en divisant le poids par la taille au carré. Le chiffre obtenu est comparé à certaines valeurs pour déterminer la corpulence de la personne (anorexique, maigre, normal, surpoids, obésité modérée, sévère ou morbide).

La décomposition de ce processus en instructions élémentaires donnerait :

- Prendre la taille et le poids
- Calculer l'IMC
- Comparer l'IMC avec les indices de corpulence
- Afficher le résultat

Dans le traitement algorithmique, nous avons besoin de variables (dans notre exemple ce sont `taille`, `poids` et `résultat`), de faire des opérations (ici `/` et `^`), et des comparaisons (conditions). Nous avons aussi besoin de *lire*, *écrire* et *enregistrer*.

Pour certains algorithmes comprenant des itérations nous avons la possibilité de faire des boucles.

Les instructions doivent être formulées dans un langage compréhensible par l'exécutant. Dans le cas d'un humain, il s'agira du langage courant (langue maternelle), L'écriture d'un algorithme dans le langage humain se nomme aussi pseudo-code.

Le pseudo-code se compose de trois blocs. Il débute toujours par un entête qui définit le nom de notre algorithme précédé du mot clé `Algorithme`, puis est suivi de la déclaration des variables qui est repéré par le mot-clé "`Variables`" (au singulier s'il n'y a qu'une seule variable à déclarer) et enfin le bloc des traitements, le corps de l'algorithme, qui commence par le mot-clé `Début` et s'achève par celui de `Fin`. Le corps peut lui même être composé de blocs, délimités par un début et une fin (tout comme le corps).

Enfin on peut y noter des commentaires qui sont marqués en début de ligne par un double-slash (`//`).

Dans notre exemple nous débuterons ainsi :

```
Algorithme IMC  // Calcul de l'indice de masse corporelle

Variables
      ...

Début
      ...
Fin
```

3. LES VARIABLES

C'est la base de tout algorithme car sans variable, il n'y a pas lieu de faire d'algorithme.

En informatique, les variables sont des symboles qui associent un nom (l'identifiant) à une valeur. Les variables peuvent changer de valeur au cours du temps. En algorithmie la variable possède deux caractèristiques :

- son **nom** c'est-à-dire sous quel nom unique est déclarée la variable ;
- son **type**, c'est la convention d'interprétation de la séquence de bits qui constitue la variable. En des mots plus simples cela veut dire que l'on renseigne sur la nature de la variable (un texte, un nombre, ou un état).

La première des choses à faire est donc de déclarer les variables que nous allons utiliser. C'est à dire que l'on informe le système que nous allons utiliser un emplacement de la mémoire pour stocker une valeur.

Par exemple dans le programme d'IMC, les variables sont la taille, le poids et le résultat. La déclaration se fait ainsi :

```
Variables
      Identifiant : type
```

3.1 NATURE DES VARIABLES

Une variable peut être

- un nombre : soit un entier, soit un réel.
- un texte : un caractère, un mot, un texte.
- un booléen : soit vrai, soit faux.

3.1.1 Les nombres

3.1.1.1 Type entier

Ce sont les chiffres sans partie décimale. Dans notre exemple, les trois variables sont de type entier.

3.1.1.2 Type réel

C'est un chiffre qui possède une partie décimale.

3.1.2 Les textes

3.1.2.1 Type caractère

Uniquement un et un seul caractère alphanumérique.

3.1.2.2 Type texte

Ce type peut prendre n'importe quel caractère alphanumérique.

3.1.3 État

3.1.3.1 Type booléen

C'est un type logique indiquant un état : soit vrai soit faux. C'est à dire soit 1 soit 0.

3.1.4 Listes et tableaux

Il existe des variables un peu spéciales : les tableaux. Ils permettent d'associer dans une même variable des valeurs de même type qui sont indexées par des entiers. Ils peuvent être à une ou plusieurs dimensions qu'on peut nommer *listes*.

3.2 DECLARER UNE VARIABLE

La déclaration est la première chose à faire dans un algorithme. Elle se trouve donc au tout début dans un bloc bien défini précédé du mot-clé "`Variables`"

Dans notre exemple la déclaration des variables ce fait ainsi :

```
Variables
     taille : entier
     poids : entier
     résultat : entier
```

Elle pourrait se faire avec chaque identifiant à la suite séparé par une virgule.

```
Variables
     taille, poids, résultat : entiers
```

3.2.1 Déclarer les tableaux

Les tableaux se déclarent en mentionnant la taille (du début à la fin) du tableau et son type comme ce qui suit :

```
Variable
      tab : tableau de 1 à 10 de entier     // Déclare un tableau de 10 entiers
      tab1 : tableau de 1 à 5 de texte      // Déclare un tableau de 5 textes
```

Pour les tableaux à plusieurs dimensions une notation entre crochets avec les indices de départ et de fin ainsi que le type est beaucoup plus lisible :

```
      tab : tableau [1..2][1..3] de entier // Tableau de 2 lignes sur 3 colonnes de type entier
```

3.3 AFFECTER UNE VARIABLE

L'affectation est l'action de donner une valeur à une variable. Si la variable avait déjà une valeur, celle-ci est remplacée par la nouvelle valeur.

Par convention l'affectation s'écrit avec une flèche qui pointe sur la variable à laquelle on donne une valeur. Mais on peut aussi avoir un texte mentionnant qu'une certaine variable prend une certaine valeur, mais aussi directement "*deux points égal*" (`:=`). Parfois il est possible de trouver la formule `assigner` avec entre parenthèse le premier paramètre la variable affectée et en second paramètre la valeur à injecter.

```
taille <- 175
poids prend la valeur de 73
x := 3
assigner(t, "mon texte")
```

On affecte `73` à `poids`, `175` à `taille`, `3` à `x` et `"mon texte"` à `t`.
C'est à dire que `taille` vaut `175`, `poids` vaut `73`, `x 3` et `t "mon texte"` même si auparavant ces variables avaient déjà une valeur.

Pour un type `réel` comme `PI`, l'affectation sera la suivante :

```
PI <- 3.14159265358
```

Si la valeur n'est pas décimale mais qu'on veut tout de même l'injecter :

```
Variable
      R : réel

R <- 1.0
```

On peut affecter la valeur d'une autre variable que si les deux variables sont de **même type**.

Par exemple :

```
Variables
      a, b : entier
      PI : réel
```

```
a <- b
```
est juste tandis que
```
a <- PI
```
est faux car `PI` est un nombre réel et `a` un entier.

On peut aussi affecter le résultat d'un calcul par exemple (c'est valable pour les autres types aussi) :
```
Variables
     a, b : entier

a <- 5
```
puis
```
a <- a + 1
```

Cela est possible parce que la **priorité de l'affectation** va de droite à gauche. Ainsi on évalue l'expression `a + 1` que l'on injecte à `a` qui possédait déjà une valeur (`5`).

Au moment de l'opération `a` est égal à `5`. On lui ajoute `1`. `a` devient donc égal à `6`.

Pour un caractère la valeur est notée impérativement entre de simples guillemets (`'`) bien que l'on voit aussi parfois les double guillemets (`"`) tandis que pour un texte on utilise obligatoirement les doubles guillemets.
```
caractère <- 'a'
texte <- "Ceci est un texte."
```

Evidemment si le texte est un chiffre et qu'on veut l'affecter à un entier c'est faux.
```
Variables
     t : texte
     i : entier

t <- "423"
i <- t          // FAUX
```

Pour affecter une valeur à un booléen :
```
Variable
     bool : booléen

bool <- vrai
```

Enfin pour affecté une valeur à un élément d'un tableau, il faut mentionner l'indice (**toujours entre crochets**) :
```
tab[1] <- 6     // On affecte 6 au premier élément du tableau tab
```

On peut faire une affectation globale d'un tableau en mettant toutes les valeurs entre deux crochets :

```
Variable
     tab : tableau de 1 à 5 de caractère
     lg : tableau [1..2][1..3] de entier
Début
     tab <- ['a', 'b', 'c', 'd', 'e']      // Le premier indice est égal à a, le second à b, etc...
     lg <- [('A', 'B'), ('a', 'b', 'c')]   // Les sous-tableaux (ou listes) sont entre parenthèses et séparés par une virgule.
     ...
```

De la même manière que l'affectation, pour lire un élément du tableau on mentionne son indice.

```
lire (tab[3])          // On lit le 3° élément du tableau (c)

lg[2;3] <- "6"         // On remplace la valeur de la 3° colonne de la 2° ligne par "6"
```

Noté le point-virgule pour séparer les deux indices.

3.4 OPERATIONS SUR LES VARIABLES

3.4.1 Opérations arithmétiques sur les variables

Il existe des opérateurs arithmétiques pour effectuer des opérations sur les nombres. On peut ainsi **additionner** (+), **soustraire** (-), **diviser** (/), **multiplier** (*, le x étant utilisé comme caractère) et mettre un nombre à la **puissance** (^).

```
a <- b^3                    // a prend la valeur de b au cube
a <- (a + b) / (6 * 5^2)
```

3.4.2 Concaténation de texte

Pour ajouter du texte à un autre texte ou une variable texte, l'opérateur **&** le permet :

```
Variable
     t1, t2 : textes

t1 <- "Bonjour"
t2 <- "les stagiaires"
t1 <- t1 & " " & t2   // Ce qui nous donne "Bonjour les stagiaires".
```

4. LES BOUCLES

Les boucles permettent d'éviter une répétition des instructions.

Par exemple lire 5 fois un texte sans les boucles s'écrirait ainsi :

```
lire le texte
lire le texte
lire le texte
lire le texte
lire le texte
```

Mais heureusement l'algorithmie et la programmation offre la solution avec des boucles !

Il en existe deux :

`TANT QUE`

Par exemple `TANT QUE` le gâteau n'est pas cuit, le four chauffe.

Cela implique de surveiller le gâteau et non le temps de cuisson.

`POUR`

Par exemple Faire chauffer le four pendant 30 minutes.

Cela implique que l'on surveille le temps. On fait donc un décompte du temps.

4.1 La boucle POUR

Le décompte du temps de notre exemple se formalise en pseudo-code de la manière suivante :

```
POUR i = 1 jusqu'à 30 minutes
     Chauffer le four
FIN POUR
```

On remarque l'utilisation d'une nouvelle variable : `i (: entier)`, qui nous permet de faire le décompte du nombre de fois qu'on a besoin. D'autant plus qu'à chaque tour `i` s'incrémente de `1` (par défaut parce qu'on a la possibilité d'utilisé un pas différent) :

```
POUR i = 1 pas de 2 jusqu'à 20
     instructions
FIN POUR
```

Conventionnellement la boucle se termine obligatoirement par `FIN POUR`.

La boucle `POUR` permet généralement d'initialiser les tableaux :

```
POUR i = 1 jusqu'à <taille du tableau>
     tab[i] <- 0
FIN POUR
```

Comme on peut imbriquer les boucles, on peut de la même manière initialiser un tableau à plusieurs dimensions :

```
POUR i = 1 jusqu'à <taille de la première liste>
     POUR j = 1 jusqu'à <taille de la seconde liste>
          tab[i][j] <- 0
     FIN POU
FIN POUR
```

4.2 La boucle TANT QUE

Cette boucle implique une condition.

```
TANT QUE le gâteau n'est pas cuit
     Faire quelque chose
FIN TANT QUE
```

Ici aussi la boucle se termine par `Fin tant que`.
Dans cet exemple la condition est l'**état** du gâteau. Il nous faut donc une variable pour stocker l'état du gâteau. Cet état peut être soit "oui il est cuit" (= vrai) ou "non il n'est toujours pas cuit" (= faux).

C'est donc un type booléen qui sera initialisé au début à faux (car le gâteau n'est pas cuit).

```
Variable
      cuit : booléen

cuit <- faux
```

Ce qui permet d'avoir la boucle suivante

```
TANT QUE cuit = faux
      cuire()
FIN TANT QUE
```

On sort automatiquement de la boucle lorsque le gâteau est cuit, et on passe aux instructions suivantes.

`cuire` prend des parenthèse parce que cette instruction n'est pas un des traitements basiques des algorithmes (`lire`, `afficher`, `enregistrer`). `cuire` peut donc être traiter dans un autre bloc algorithmique.

5. LES CONDITIONS

Comme dans le langage courant la condition est "**si**". *Si le gâteau est cuit alors on le sort du four*. Cette condition "élémentaire" se traduit en pseudo-code comme suit :

```
SI condition ALORS
      instructions
FIN SI                // La condition se termine toujours par FIN SI
```

On peut cependant avoir envie de gérer la condition qui n'est pas remplie comme *Si le gâteau est cuit alors on le sort du four sinon on continue à le cuire*. Ce qui se traduit ainsi :

```
SI cuit = faux ALORS
      Continuer à cuire le gâteau
SINON
      Sortir le gâteau du four
FIN SI
```

On peut avoir des conditions plus évoluées comme par exemple *Si la batterie de mon ordinateur portable est inférieur à 20% de sa charge alors afficher une icone rouge, si elle est supérieur à 20% mais inférieur à 80% on affiche une icone jaune et si elle est supérieure à 80% on affiche une icone verte*. Nous avons ici trois conditions. Toutefois `SI` ne gère que deux conditions. Il va donc falloir imbriquer un deuxième test dans le premier.

```
SI charge < 20% ALORS
      afficher icone rouge
SINON           // On ajoute notre seconde condition
```

```
     SI charge < 80% ALORS
          afficher icone jaune
     SINON      // La troisième condition
          afficher icone verte
     FIN SI
FIN SI
```

5.1 LES OPERATEURS CONDITIONNELS

Afin de pouvoir évaluer les conditions le pseudo-code nous permet d'utiliser les opérateurs suivant :

Opérateur	Description
<	inférieur
<=	inférieur ou égal
=	égal
<>	différent
>=	supérieur ou égal
>	supérieur

6. EXERCICE

On veut déduire l'âge d'une personne.

L'utilisateur va entrer l'âge de la personne tant qu'il ne correspond pas à l'âge de celle-ci. Si l'utilisateur trouve l'âge on affiche un message d'erreur.

6.1 DEROULE DU PROGRAMME :

1. L'âge est entré "*en dur*" dans l'ordinateur.
2. L'ordinateur affiche "*Quel âge a Momo ?*"
3. L'ordinateur est en attente d'une valeur.
4. Lorsque la valeur est entrée, l'ordinateur compare l'âge entré avec l'âge prédéfini.
5. 3 cas possible :
    a. L'âge **correspond**

       Le programme affiche un message de félicitation et le programme s'arrête.
    b. L'âge entré est **inférieur**

       Le programme affiche que l'âge est inférieur et il attend une nouvelle valeur.
    c. L'âge entré est **supérieur**

       Le programme affiche que l'âge est supérieur et il attend une nouvelle valeur.

6.2 VARIABLES

1. L'âge de la personne

   Entier

   âge : entier
2. L'âge que donne l'utilisateur

   Entier

   i : entier

6.3 INTERACTIONS

1. Affichage du message "*Quel âge a Momo ?*"
2. Lire la valeur entrée au clavier.

   Mettre la valeur dans `i`

   `i <-` valeur entrée au clavier
3. Comparaison des valeurs.

```
Si ... alors
      …
Fin Si
```

4. Affichage du résultat de la comparaison

6.4 PSEUDO-CODE

1. Déclaration des variables

```
Variables
      âge, i : entiers
```

2. Affectation de l'âge de Momo

```
âge <- 62
```

3. Affichage

```
Afficher "Quel âge a Momo ?"
```

4. Lire la valeur entrée par l'utilisateur

```
Lire i
```

5. Comparaison

```
Si âge = i alors afficher "Félicitation"
Sinon
      Si age > i alors afficher "Momo est plus vieux"
      Sinon afficher "Momo est plus jeune"
Fin Si
```

Cependant l'utilisateur entre des valeurs **TANT QUE** la valeur entrée est fausse. La comparaison se trouve alors dans une boucle `TANT QUE`.

Ce qui fait :

3.

```
TANT QUE i différent de âge
```

4. Affichage

```
      Afficher "Quel âge a Momo ?"
```

5. Lire la valeur entrée par l'utilisateur

```
      Lire i
```

6. Comparer

```
      Si âge > i alors
            afficher "Momo est plus vieux"
      Sinon
            afficher "Momo est plus jeune"
      Fin Si
```

7. Fin de la boucle

```
Fin TANT QUE
Affichage "Félicitations"
```

6.5 AVENANT A L'EXERCICE

On veut compter le nombre de fois que l'utilisateur entre une valeur.

On doit donc créer une nouvelle variable :

```
compteur : entier
```

Que l'on initialise à 0

```
compteur <- 0
```

Dans la boucle `TANT QUE` on incrémente `compteur` de 1 unité.

```
compteur <- compteur + 1
```

L'affichage final devient

```
"Félicitations (" & compteur & " essais)"
```

CREDITS
Math rix
ŒUVRE COLLECTIVE DE l'AFPA
Sous le pilotage de la DIIP et du centre d'ingénierie sectoriel Tertiaire-Services

Equipe de conception (IF, formateur, mediatiseur)
Ludovic Domenici - Formateur informatique
Ch. Perrachon – Ingénieure de formation

Date de mise à jour : 04/12/18