Chapitre 1 - Rappel sur l'algorithmique

I

L'objectif de ce premier chapitre de ce cours de rappeler rapidement les concepts de base d'algorithmique déjà vus en première année. Nous allons d'abord commencer par la définition de ce que c'est un algorithme, puis nous allons présenter les instructions de base qu'on a le droit d'utiliser pour concevoir un algorithme permettant de résoudre un problème donné. Finalement, il faut savoir qu'un algorithme n'est pas un code exécutable, mais juste une description détaillée des étapes de résolution d'un problème pouvant facilement être traduite en programme écrit dans n'importe quel langage de programmation, dans notre cas nous allons utiliser Matlab.

1. Qu'est ce qu'un algorithme

Définition : Algorithme

Un algorithme est une séquence d'étapes obéissant à un enchaînement bien déterminé, permettant de résoudre un problème spécifique. Un algorithme est correcte s'il est capable de résoudre le problème pour chaque instance donnée.

Définition : Problème

Un problème est un questionnement nécessitant une solution.

Complément : Résoudre un problème

La résolution d'un problème est de concevoir une méthode de raisonnement qui permet de construire en un nombre fini d'étapes élémentaires l'énoncé de la solution. ce qu'on peut appeler autrement *trouver un algorithme*.

Fondamental : Étape élémentaire

Chacune des étapes élémentaires qui composent un algorithme doit satisfaire les deux critères suivants :

- *non-ambigue* : c'est à dire définie d'une façon rigoureuse et ne peut pas être ouverte à différentes interprétations ;
- *effective* : pouvant être effectivement réalisée par une machine.

Définition : Programme

Un programme destiné à être exécuté par un ordinateur est la description d'un algorithme dans un langage accepté par cette machine (par exemple : un programme écrit en langage Matlab).

2. Structure de données

Définition

Une structure de données est un ensemble organisé d'informations reliées, ces informations peuvent être traitées collectivement ou individuellement.

Exemple : Les tableaux

L'exemple le plus simple et le plus répandu d'une structure de données est les tableaux monodimensionnels (appelés aussi vecteurs). Un tableau est constitué d'un certain nombre de composantes de même type. On peut effectuer des opérations sur chaque composante prise individuellement mais on dispose aussi d'opérations globales portant sur le tableau considéré comme un seul objet.

Attention

Une structure de données est caractérisée par ses composantes et leur arrangement mais surtout par son mode de traitement. Ainsi deux structures ayant les mêmes composantes et les mêmes arrangements peuvent être considérées comme différentes si leurs modes d'exploitation sont différents.

3. Structure d'un algorithme

Fondamental

Un algorithme se compose de trois parties fondamentales : l'*entête*, la *partie déclaration* et le *corps de l'algorithme*.

- Dans l'entête, on donne un nom à l'algorithme en utilisant la syntaxe suivante : `algorithme nom_algorithme;` il est recommandé que le nom de l'algorithme soit significatif du problème qu'il résout :
  - exemple d'un identifiant significatif : `algorithme somme;`
  - exemple d'un identifiant non-significatif : `algorithme Alg01;`
- Dans la partie déclaration, il faut définir toutes les variables qui seront utilisées dans le corps de l'algorithme en utilisant la syntaxe suivante : `var nom_variable : type;` il est recommandé que le nom d'une variable soit significatif de son rôle dans le corps de l'algorithme pour faciliter la lecture de ce dernier.
- Le corps de l'algorithme consiste en une séquence d'instruction placé entre les deux mots clés `début` et `fin`.

Remarque : La partie déclaration et types de variables

Vu que nous allons utiliser le langage Matlab (qui n'oblige pas à déclarer des variables avant de les initialiser, et qui ne limite pas une variable à un seul type de données) pour traduire les algorithmes en programmes, nous allons ignorer la partie déclaration pour garder une cohérence entre les algorithmes et leurs programmes correspondants. Nous supposons aussi qu'une variable peut changer de type (on peut affecter des valeurs de types différents à la même variable).

3.1. Types de données

La notion de type est très importantes en algorithmique (ainsi qu'on programmation) que toute valeur affacté à une variable doit forcément avoir un type. Nous distinguons quatre types élémentaires :

- le type `entier` : utilisé pour stocker des valeurs entières positives ou négatives ;
- le type `réel` : utilisé pour stocker des nombres réels (à virgule) ;
- le type `caractère` : utilisé pour stocker des caractères ;
- le type `booléen` : utilisé pour stocker des valeur binaire ou logiques (0/1 ou vrai/faux).

Remarque : Retour sur les structures de données

Les types de bases peuvent être utilisés comme briques de base pour créer des types plus complexes, c'est ce que nous appelons *les structures de données*. Les tableaux sont la structure de données la plus importante pour la suite de ce cours, on peut définir des tableaux d'entiers, de réels, de caractères (ou aussi chaînes de caractères) ou de booléen.

On peut aussi utiliser les structure de données qu'on a personnalisées comme composantes d'autres structures encore plus complexes. Prenons l'exemple d'un tableau de tableau d'entiers, aussi appelé tableau bi-dimensionnel ou matrice.

3.2. Variables

Les variables sont utilisés dans un algorithme pour pouvoir manipuler des données, chaque variable doit posséder :

- *Un nom* : aussi appelé un identificateur, ce dernier doit être unique pour permettre à l'algorithme de reconnaître d'une façon précise quelle donnée manipuler.
- *Une valeur* : qui évolue au cours d'évolution de l'algorithme. Cette valeur peut être lue ou modifiée en évoquant l'identifiant de la variable correspondante.

Cette notion (variable) est fondamentale, parce qu'elle permet de manipuler des valeurs (numériques ou non) et est à la base de la plupart des calculs et traitements que l'on peut réaliser avec un langage de programmation.

Remarque : Retour sur les structures de données

Les types de bases peuvent être utilisés comme briques de base pour créer des types plus complexes, c'est ce que nous appelons *les structures de données*. Les tableaux sont la structure de données la plus importante pour la suite de ce cours, on peut définir des tableaux d'entiers, de réels, de caractères (ou aussi chaînes de caractères) ou de booléen.

On peut aussi utiliser les structure de données qu'on a personnalisées comme composantes d'autres structures encore plus complexes. Prenons l'exemple d'un tableau de tableau d'entiers, aussi appelé tableau bi-dimensionnel ou matrice.

3.3. Le corps de l'algorithme

C'est dans cette partie qu'on décrit les étapes élémentaires à suivre pour résoudre le problème. Ces étapes élémentaires sont aussi appelées *des instructions*. Nous expliquons de quoi s'agit le terme instructions avec plus de détails dans la prochaine section.

4. Instructions de base et opérations

Les instructions décrivent les actions à effectuer par l'algorithme pour résoudre le problème. Chaque deux instructions consécutives sont séparées par un point-virgule. Nous allons diviser les instructions en plusieurs catégories.

4.1. Les instructions de lecture et écriture

Il s'agit des deux instruction `lire` et `écrire`. L'instruction `lire` permet d'initialiser une variable à partir d'une saisie faite au clavier, alors que l'instruction `écrire` comme son nom l'indique sert à afficher du texte ou la valeur d'une variable.

Complément : L'instruction de lecture

Pour initialiser une variable saisie par l'utilisateur, on utilise la fonction `lire`, suivi du nom de la variable que l'on veut saisir entre deux parenthèses. L'instruction `lire` a pour seul effet d'attendre que l'utilisateur saisisse une valeur au clavier et la valide en appuyant sur la touche entrée (ou enter dans un clavier anglais), aucun message ne s'affiche pour indiquer à l'utilisateur ce qu'il doit faire, il est donc recommandé d'utiliser la fonction `écrire` pour afficher un message à l'utilisateur lui expliquant qu'il doit saisir une valeur.

Complément : L'instruction d'écriture

Pour afficher un élément à l'écran, on utilise la fonction `écrire`, suivi entre parenthèses de l'élément à faire apparaître . Cet élément est soit du texte brut écrit entre doubles guillemets, soit une variable dont on veut afficher la valeur ou une expression à évaluer avant d'afficher sa valeur. Dans le cas de texte brut, ce dernier apparaît tel quel à l'écran. Dans le cas d'une expression, c'est le résultat du calcul de cette expression qui est affiché.

Exemple : Utilisation des instructions lire et écrire

```
1 algorithme lire_et_écrire;
2 début
3 écrire("Veuillez saisir la valeur de a : ");
4 lire(a);
5 écrire(a);
6 fin.
```

4.2. L'affectation

L'opération d'affectation permet de donner (ou d'affecter) une valeur à une variable en utilisant la syntaxe suivante :

```
variable  expression;
```

La valeur de la variable à gauche de l'opération d'affectation est mise à jour par le résultat d'évaluation de l'expression à droite. Cette expression peut être soit une valeur constante, soit une variable, une expression arithmétique ou autre types d'expressions que nous allons voir par la suite.

Exemple : Utilisation de l'affectation

```
1 algorithme affectation;
2 début
3 a ← 1;
4 b ← 2;
5 c ← a + 3; % c vaut 4
6 d ← c * b; % d vaut 8
7 fin.
```

4.3. Conditions et boucles

Ces structures permettent au programme de ne pas être purement séquentiel. Suite à la complexité de ces structures nous allons consacrer la prochaine section pour les expliquer avec plus de détails.

5. Les structures de contrôle

Les structures de contrôle sont utiles dans le cas ou l'avancement de l'exécution d'un algorithme peut prendre des chemins différents selon l'évolution de ses variables. L'exemple le plus simple d'un problème mathématique qui nécessite l'utilisation des contrôle est la résolution d'une équation du second degré. Cette dernière a deux, une ou aucune solution en fonction de la valeur de `delta`. Donc l'algorithme qui résout ce problème devra adapter son comportement en fonction des valeurs prises par certaines variables.

5.1. Exécution conditionnelle d'un bloc d'instructions

Définition : Condition

Une condition est une expression dont le résultat est une valeur logique, pour cela nous utilisons des opérateurs logiques. Les opérateurs de comparaison (<, >, =, ≤, ≥, ≠) sont les plus utilisés pour tester des conditions (par exemple : si `a > b`).

Complément : Opérateurs logiques de base

Les trois opérateurs logiques de base (`et, ou, non`) sont utilisés pour mieux manipuler les conditions (par exemple : tester si plusieurs conditions sont satisfaites).

- L'opérateur `et : cond1 et cond2` retourne la valeur logique vrai si et seulement si la condition `cond1` et la condition `cond2` retourne toutes les deux la valeur logique `vrai`, sinon le résultat retourné est la valeur logique `faux`.
- L'opérateur `ou : cond1 ou cond2` retourne la valeur logique vrai si et seulement si au moins l'une des deux conditions `cond1` et `cond2` retourne la valeur logique `vrai`, sinon le résultat retourné est la valeur logique `faux`.
- L'opérateur `non : non cond` retourne la valeur logique `vrai` si et seulement si `cond` retourne la valeur logique `faux`.

La structure `si ... alors` permet d'exécuter des instructions en fonction de la valeur d'une condition (ou le résultat d'un test). La syntaxe utilisée est la suivante :

```
si cond alors
début
   instruction_1;
   instruction_2;
   ..........
   instruction_n;
fin;
```

Cette structure fonctionne de la manière suivante :

- si la condition `cond` est `vraie`, les instructions écrites entre `début` et `fin` sont exécutées ;
- alors que lorsque la condition `cond` est fausse les instructions ne sont pas exécutées.

Dans la suite nous appelons une séquence d'instructions entre les deux mots clés début et fin un bloc d'instructions.

Exemple : Utilisation de la structure "si ... alors"

Dans cet exemple nous allons écrire un algorithme qui attend que l'utilisateur saisisse deux nombres : `a` et `b`, puis il calcule et affiche le résultat de la division de `a` par `b`. Évidement la division par zéro ne peut pas être effectuée, donc si l'utilisateur donne la valeur `0` à la variable `b` l'opération de division n'est pas exécutée et le résultat n'est pas affiché.

```
 1 algorithme division;
 2 début
 3 lire(a);
 4 lire(b);
 5 si b ≠ 0 alors
 6    début
 7    c ← a / b;
 8    écrire(c);
 9    fin; % fin pour la condition avec un ';'
10 fin.    % fin de l'algorithme avec un '.'
```

Complément : si ... alors ... sinon

Il arrive souvent qu'un programme doive exécuter un bloc d'instructions si une condition est vraie, et un autre bloc si cette même condition est fausse. Plutôt que de tester une condition puis son contraire, il est possible d'utiliser la structure `si ... alors ... sinon`, dont la syntaxe est la suivante :

```
si cond alors
bloc_instructions_1;
sinon
bloc_instructions_2;
```

Cette structure fonctionne de la manière suivante :

- si la condition `cond` est vraie, alors le premier bloc d'instructions est exécuté ;
- sinon, le deuxième bloc d'instructions est exécuté.

Pour clarifier encore plus le fonctionnement de cette structure, nous allons donner un exemple en changeant l'algorithme précédent (division de `a` par `b`).

Exemple : Utilisation de la structure "si ... alors ... sinon"

```
1 algorithme division;
2 début
3 lire(a);
4 lire(b);
5 si b ≠ 0 alors
6    début
7    c ← a / b;
8    écrire(c);
9    fin;
10 sinon
11    début
12    écrire("Il n'est pas possible de diviser par zéro");
13    fin;
14 fin.
```

Dans l'exemple précédent la division par b n'est effectuée que lorsque la valeur de b est différente de zéro. Dans le cas inverse un message est affiché pour indiquer qu'il est impossible de diviser par zéro.

5.2. Les structures itératives

Certains algorithmes nécessitent la répétition de certains instructions plusieurs fois avant d'obtenir le résultat voulu. Cette répétition est réalisée en utilisant une structure de contrôle de type itératif, nommée boucle.

Définition : La boucle "tant que ... faire"

Sa syntaxe de la boucle `tant que` est la suivante :

```
tant que cond faire
   bloc_instructions;
```

Lorsque l'ordinateur rencontre cette structure, il procède de la manière suivante :

1. La condition est testée.
2. Si la condition est vraie, l'instruction ou les instructions du bloc sont exécutées, après on revient à l'étape 1 (tester la condition).
3. Si la condition est fausse, l'instruction ou les instructions du bloc ne sont pas exécutées et on passe aux instructions suivantes (après la boucle).

Exemple : Utilisation de la structure "tant que ... faire"

Dans cet exemple nous allons écrire un algorithme qui affiche tous les nombres entiers positifs entre 0 et 10 (inclus).

```
1 algorithme boucleTQ;
2 début
3 x ← 0;
4 tant que x ≤ 10 faire
```

```
5    début
6    écrire(x);
7    x ← x + 1;
8    fin;
9 fin.
```

Complément : La boucle "pour ... faire"

Lorsque le nombre d'itérations dont un bloc d'instructions doit être exécuté est connu à l'avance, la boucle pour est préférable a la boucle tant que. L'usage principal de la boucle pour est de faire la gestion d'un compteur qui évolue d'une valeur à une autre. La boucle pour est utilisée suivant la syntaxe suivante :

```
pour compteur allant de valeur1 à valeur2 faire
   bloc_instructions;
```

Lorsque l'ordinateur rencontre cette structure, il procède de la manière suivante :

1. la variable, jouant le rôle de compteur, est initialisée à la valeur1 ;
2. l'ordinateur teste si la variable est inférieure ou égale à la valeur2 :
    - si c'est le cas, l'instruction ou le bloc d'instruction est exécuté, la variable jouant le rôle de compteur est augmentée de 1, et puis on retourne à l'étape 2, et non à l'étape 1 qui initialise la variable ;
    - sinon, le bloc d'instructions n'est pas exécuté, et l'ordinateur passe aux instructions après la boucle

Exemple : Utilisation de la structure "pour ... faire"

Dans cet exemple nous allons réécrire l'algorithme précédent (affichage des nombres entre 0 et 10) en utilisant la boucle pour au lier de la boucle `tant que`.

```
1 algorithme bouclePour;
2 début
3 pour x allant de 0 à 10 faire
4    début
5    écrire(x);
6    fin;
7 fin.
```

Remarque

En réalité, les deux boucles `pour` et `tant que` peuvent être utilisées de façons différentes pour résoudre le même problème. Toute boucle `pour` peut être reformulée et écrite sous la forme d'une boucle `tant que`.

Conseil

Il est recommandé d'utiliser la boucle `pour` si le nombre d'itérations est connu à l'avance (avant d'entrer dans la boucle) et la boucle `tant que` lorsque le nombre d'itérations n'est pas connu.

6. Fonctions

Le rôle d'une fonction est de regrouper des instructions qui doivent être exécutées d'une manière répétitive dans ce qu'on appelle un sous programme. Une fonctions peut avoir plusieurs entrées et une sortie. Les entrées de la fonction sont les valeurs qu'on lui fournit pour qu'elle puisse les traiter et donner un résultat. La sortie est une valeur retourner par la fonction. cette valeur peut être de n'importe quelle type (types prédéfinis, tableaux ou autres structures de données).

Attention

Le type des entrées et de la sortie de la fonction doivent être définis, mais comme nous avons signalé dans les précédentes sections, nous supposons qu'on peut affecter à une variable des valeurs de différents types. Autrement dit, une variable dans notre cas (comme dans plusieurs langage de programmation, Matlab inclu) une variable peut changer de type. Nous partons du même principe pour les fonctions. Donc le type de résultat retourné par une fonction peut changer pour chaque appel.

Complément : Définir une fonction

Une fonction doit être définie avant d'être utilisée, c'est-à-dire que l'on doit indiquer son entête (nom de la fonction et ses paramètres ou entrées) et son corps (les instructions qui la composent). La syntaxe de l'entête est la suivante :

```
fonction nom_fonction(entrée1, entrée2, ........., entréen);
```

Alors que le corps est un bloc d'instructions (suite d'instructions délimitées par les deux mots clés `début` et `fin`).

Fondamental : L'instruction de retour

Les fonctions utilisent une instruction spécifique permettant de communique sa sortie à l'algorithme qui l'utilise. La sortie d'une fonction (il y en a au maximum une) est en fait une valeur (numérique ou autre) que celle-ci fournit. Il nous faudrait donc une instruction dont l'effet serait : « répondre à l'algorithme appelant la valeur qu'il demande ». Cette instruction existe, son nom est retourner. la syntaxe de cette instruction de retour est la suivante.

```
retourner expression;
```

Attention

L'instruction retourner est toujours la dernière instruction exécutée dans une fonction. Toute instruction placée après l'instruction `retourner` est ignorée.

Exemple

Une analogie avec une fonction mathématique permet d'illustrer clairement les notions d'entrées et de sorties. Soit la fonction suivante définie avec le formalisme des mathématiques :

$$\begin{aligned} F &: \mathbb{R}\times\mathbb{R}\to\mathbb{R} \\ x,y &\to x^2+y^2+2xy \end{aligned} \qquad (1)$$

```
fonction identitéRemarquable1(x, y);
début
res = x * x + y * y + 2 * x * y;
retourner res;
fin;
```

Dans l'exemple précédent, la variable `res` est utilisée pour calculer la valeur de l'expression évaluée `(x² + y² + 2xy)`. Puis l'instruction suivante retourne la valeur enregistrée dans la variable `res`.

Remarque : Fonction sans valeur retournée

Il est possible qu'une fonction n'ait aucune valeur à retourner. Dans ce cas il n'est pas indispensable d'utiliser l'instruction retourner. Les fonctions qui ne retournent aucune valeur sont très souvent appelées des *procédures.*

6.1. Appels de fonctions

L'appel d'une fonction correspond à une demande de son utilisation. Afin d'appeler une fonction, on doit préciser son nom, ainsi que les valeurs que l'on fournit pour les entrées (paramètres ou arguments). La fonction appelée est en charge de fournir la sortie. La syntaxe suivante est utilisée pour faire appel à une fonction :

```
nomFonction(param1, param2, ...., paramn);
```

Exemple : Trouver la valeur maximale dans un tableau

Pour expliquer comment marche l'appel aux fonctions nous allons utiliser un exemple ou on cherche la valeur maximale dans un tableau unidimensionnel contenant des valeurs numériques. Nous allons premièrement définir une fonction max qui prend comme paramètres deux valeurs numériques et retourne la plus grande valeur des deux. Puis nous utilisons la fonction max pour définir la fonction maxTab qui prend comme paramètres un tableau contenant des valeurs numériques et un entier indiquant la taille de ce tableau, et qui retourne comme résultat la valeur maximale dans le tableau.

Complément : Définir la fonction max

```
fonction max(v1, v2);
début
si v1 > v2 alors
   début
   retourner v1;
   fin;
sinon
   début
   retourner v2;
   fin;
fin;
```

L'idée de la fonction max est de tester si le premier argument est supérieur au deuxième avant de décider quelle valeur retourner. Si la valeur du test est vraie elle retourne le premier argument, sinon le deuxième argument est retourné.

Complément : Définir la fonction maxTab

```
fonction maxTab(T, n);
début;
```

```
vMax = T[1];
pour i allant de 2 à n faire
   début
   vMax = max(vMax, T[i]);
   fin;
retourner vMax;
fin;
```

La première instruction de la fonction `maxTab` déclare une variable `vMax` et l'initialise avec la valeur dans la première case du tableau (`T[1]`). Ensuite toutes les valeurs contenues dans le tableau `T` sont parcourues et comparées à la variable `vMax`, la valeur maximale est gardée jusqu'à la fin de la boucle et retournée en utilisant l'instruction `retourner`.

7. Tableaux

Un tableau permet de regrouper plusieurs valeurs dans une seule variable, au sein de laquelle chaque valeur sera désignée par un numéro. En d'autres termes, un ensemble de valeurs portant le même nom (ou identifiant) et repérées par un nombre (appelé indice) s'appelle un tableau.

7.1. Manipuler les tableaux

Les instructions de base manipulent les tableaux élément par élément. Pour accéder à un élément du tableau il faut utiliser l'identifiant du tableau auquel il appartient suivi par l'indice de l'élément entre crochets. Les indice du tableau sont numérotés à partir de 1 jusqu'à la taille du tableau. L'exemple suivant montre comment lire ou écrire les élément d'un tableau.

Exemple : Manipuler les éléments d'un tableau

```
lire(A[1]);      % lire l'élément avec l'indice 1 du tableau A.
écrire(A[1]);    % afficher l'élément avec l'indice 1 du tableau A.
A[2] ← A[1] * 2; % Utiliser les éléments d'un tableau dans une affectation
```

Remarque

Il ne faut pas confondre l'indice d'un élément du tableau avec la valeur de cet élément. L'indice peut être considérer comme l'adresse fixe d'un élément alors que sa valeur est accessible via cet indice et peut changer au fur et à mesure que l'algorithme avance.

7.2. Tableaux et boucles

Les boucles sont extrêmement utiles pour les algorithmes associés aux tableaux. En effet, de nombreux algorithmes nécessitent de parcourir les éléments d'un tableau dans un certain ordre, le plus souvent dans le sens des indices croissant. Le traitement de chacun des éléments étant souvent le même, seule la valeur de l'indice est amenée à changer. Une boucle est donc parfaitement adaptée à ce genre de traitements.