Chapitre 1 : Rappels sur les Bases de l'Algorithmique

I.1 Définitions

Algorithme :

Voici quelques définitions :

a. Un algorithme est une procédure de calcul bien définie qui prend en entrée une valeur, ou un ensemble de valeurs et donne en sortie une valeur, ou un ensemble de valeurs. Un algorithme est donc une séquence d'étapes de calcul qui transforment l'entrée en sortie (Cormen, Leiserson, Rivert)

b. Un algorithme est un ensemble d'opérations de calcul élémentaires, organisé selon des règles précises dans le but de résoudre un problème donné. Pour chaque donnée du problème, il retourne une réponse après un nombre fini d'opérations. (Beauquier, Berstel, Chrétienne)

c. Un algorithme est une spécification d'un schéma de calcul sous forme d'une suite finie d'opérations élémentaires obéissant à un enchaînement déterminé. (Al Khawarizmi, Bagdad IXè siècle. *Encyclopedia Universalis)*

Exemple : Trouver son chemin, extrait d'un dialogue entre un touriste égaré et un autochtone.
– Pourriez-vous m'indiquer le chemin de la gare, s'il vous plait ?
– Oui bien sûr : vous allez tout droit jusqu'au prochain carrefour, vous prenez à gauche au carrefour et ensuite la troisième à droite, et vous verrez la gare juste en face de vous.
– Merci.

Dans ce dialogue, la réponse de l'autochtone est la description d'une suite ordonnée d'instructions (allez tout droit, prenez à gauche, prenez la troisième à droite) qui manipulent des données (carrefour, rues) pour réaliser la tâche désirée (aller à la gare).
Chacun de nous a déjà été confronté à ce genre de situation et donc, consciemment ou non, a déjà construit un algorithme dans sa tête (c.-à-d. définir la suite d'instructions pour réaliser une tâche). Mais quand on définit un algorithme, celui-ci ne doit contenir que des instructions compréhensibles par celui qui devra l'exécuter (l'humain ou l'ordinateur).

Dans ce cours, nous devrons apprendre à définir des algorithmes pour qu'ils soient compréhensibles — et donc exécutables — par un ordinateur.

Définition de l'algorithmique :

L'algorithmique est la science des algorithmes. L'algorithmique s'intéresse à l'art de construire des algorithmes ainsi qu'à caractériser leur validité, leur robustesse, leur réutilisabilité, leur complexité et leur efficacité.

L'algorithmique permet ainsi de passer d'un problème à résoudre à un algorithme qui décrit la démarche de résolution du problème.

Définition de la programmation :

L'objet et l'intérêt de la programmation est de permettre de spécifier à une machine un travail à effectuer de façon automatique. La programmation a pour rôle de traduire l'algorithme dans un langage «compréhensible» par l'ordinateur, on obtient ainsi un programme.

Définition de programme :

Il faut en général fournir à la machine les valeurs des paramètres que l'on appelle les données. Ensuite la machine effectue des opérations sur ces données en suivant un schéma de fonctionnement qui lui aura été précisé, c'est ce que l'on appelle programme. Enfin tout ceci n'a d'intérêt que dans la mesure où la machine fournit des résultats.

Données ⟶ Programme ⟶ Résultats

La résolution d'un problème :

La résolution d'un problème passe donc par les étapes suivantes :

- Problème à résoudre
- Enoncé clair et complet délimitant précisément le problème
  - ✓ que connaît-on ? (les données)
  - ✓ que cherche-t-on ? (les résultats)
- Analyser dans la langue naturelle les actions à réaliser
  - ✓ Comment passer des données aux résultats ? (traitement)
- Algorithme (Analyse qui utilise un mode d'expression précis)
- Traduction de l'algorithme dans un langage de programmation donné.

Problème ⟶ Algorithme ⟶ Programme ⟶ Résultats

Enoncé explicite du problème — *Traduction en langage de programmation*

I.2 Propriétés d'un Algorithme (Validité, Robustesse, Réutilisabilité, …)

Un algorithme doit satisfaire les propriétés suivantes :

- Il peut avoir zéro ou plusieurs quantités de données en entrée ;
- Il doit produire au moins une quantité en sortie ;
- Chacune de ses instructions doit être claire et non ambigüe ;
- Il doit terminer après un nombre finie d'étapes ;
- Chaque instruction doit être implémentable ;

Validité d'un algorithme :

La validité d'un algorithme est son aptitude à réaliser exactement la tâche pour laquelle il a été conçu.

Si l'on reprend l'exemple précédent de l'algorithme de recherche du chemin de la gare, l'étude de sa validité consiste à s'assurer qu'on arrive effectivement à la gare en exécutant scrupuleusement les instructions dans l'ordre annoncé.

Robustesse d'un algorithme :

La robustesse d'un algorithme est son aptitude à se protéger de conditions anormales d'utilisation.

Dans l'exemple précédent, la question de la robustesse de l'algorithme se pose par exemple si le chemin proposé a été pensé pour un piéton, alors que le « touriste égaré » est en voiture et que la « troisième à droite » est en sens interdit.

Réutilisabilité d'un algorithme :

La réutilisabilité d'un algorithme est son aptitude à être réutilisé pour résoudre des tâches équivalentes à celle pour laquelle il a été conçu. L'algorithme de recherche du chemin de la gare est-il réutilisable tel quel pour se rendre à la mairie? A priori non, sauf si la mairie est juste à côté de la gare.

Complexité d'un algorithme :

La complexité d'un algorithme est le nombre d'instructions élémentaires à exécuter pour réaliser la tâche pour laquelle il a été conçu. Si le « touriste égaré » est un piéton, la complexité de l'algorithme de recherche de chemin peut se compter en nombre de pas pour arriver à la gare.

Efficacité d'un algorithme :

L'efficacité d'un algorithme est son aptitude à utiliser de manière optimale les ressources du matériel qui l'exécute. N'existerait-il pas un raccourci piétonnier pour arriver plus vite à la gare ?

Les objectifs du cours

Après avoir acquis en première année les notions fondamentales de l'algorithmique, plus précisément, les instructions de base permettant de décrire les algorithmes (affectation, tests, boucles, les procédures et les fonctions qui permettent de structurer et de réutiliser les algorithmes ; le passage de paramètres, les appels de fonctions, les structures de données linéaires : tableaux et listes chainées).

En deuxième année on passe à l'étude de nouvelles structures de données (telles que les piles, les files, les arbres …). L'objectif est d'apprendre à l'étudiant à concevoir des algorithmes et à les analyser.

Sachant que la solution du même problème peut être décrite avec plusieurs algorithmes, l'étudiant doit apprendre à fournir un algorithme clair qui s'exécute en moins de temps et aussi choisir la structure de données la plus adaptée au problème. La connaissance des propriétés des structures de données permet de guider le choix des représentations dans le souci d'optimisation des performances des solutions obtenues.

I.3 Langage Algorithmique

I.3.1 Structure générale d'un algorithme

```
Algorithme <nom de l'algorithme>
Déclaration de constantes, de types et de variables
Déclaration de fonctions et/ou de procédures
Début
   <groupes d'instructions>
Fin.
```

I.3.2 Les opérateurs standard

L'affectation **:** *Nom Variable*←*Expression* **;**

Opérateurs arithmétiques

+	Addition
-	Soustraction
*	Multiplication
/	division (entière et rationnelle!)
%	modulo (reste d'une div. entière)

Opérateurs logiques

et	et logique
ou	ou logique
non ou¬	négation logique

Opérateurs de comparaison

=	égal à
≠	différent de
<, <=, >, >=	plus petit que, ...

I.3.3 Les déclarations des constantes, types et variables

Constante <nom de variable>=<valeur>
Type <nom du nouveau type>=<type>
Variable <Nom de variable> : <type>

Les types simples sont: entier, réel, caractère, booléen
Exemple:

Constante PI=3,145
Variable p : réel ; x : entier ; b : booléen
Type chaine= tableau [20] de caractères
Type info=enregistrement nom, prénom : chaine

I.3.4 Les tableaux à une dimension

Déclaration: *<Nom de variable> : **tableau** [dimension] de <type> ;*
Exemple : T : tableau [50] de entier ;

Accès aux composantes : <Nom de variable> [<position>] ;
Exemple : T[1], T[2], ... , T[50]

I.3.5 Les tableaux à deux dimensions

Déclaration :

<Nom de variable> : tableau [dimLigne][dimColonne>] ***de <type>***;

Exemple : M : tableau [10][50] de entier;

Accès aux composantes :

<Nom de variable>[<positionLigne>][<positionColonne>] ;
Exemple : M[1][1], M[1][2], ... , M[4][1], …, M[10][50]

I.3.6 Les Chaînes de caractères :

Une chaine est un tableau de caractères. Donc on défini un type chaine comme suit :

Type chaine= tableau [20] de caractères

Déclaration : *<Nom de variable> : **chaine** ;*
Exemples : Nom, Prénom : chaine ;

Les chaînes de caractères constantes :

Les chaînes de caractères constantes sont indiquées entre guillemets.

- La chaîne de caractères vide est alors : **""**
- "Bonjour" est une **chaine** de 7 caractères
- **'x'** est un **caractère** et est codé dans un octet

Remarque : Pour déterminer la taille d'une chaine on utilise la fonction ***longueur***
Exemple : ch : chaine ; lire(ch) ; n←longueur(ch) ;

I.3.7 Tableaux de chaînes de caractères

Un tableau de chaînes de caractères correspond à un tableau à deux dimensions du type **caractère**, où ***chaque ligne contient une chaîne de caractères***.

Déclaration : La déclaration **Jour : tableau [7] de chaine;** Réserve l'espace en mémoire pour 7 mots.

Accès aux chaînes : Il est possible d'accéder aux différentes ***chaînes de caractères*** d'un tableau, en indiquant simplement la ligne correspondante : Jour[3] ←"mercredi"

I.3.8 Lire et afficher des données :

L'instruction « Lire » ordonne à la machine de lire ou prendre une donnée (valeur) qu'elle va mettre dans sa mémoire.

Lire (<nom de variable>) ;
Lire (<var1>, <var2>, …) ;

L'instruction « Ecrire » permet de copier les résultats de la mémoire pour les afficher à l'écran. Elle signifie aussi ordonner à la machine d'écrire ou d'afficher le contenu d'une variable ou encore afficher un message.

Ecrire (<nom de variable>) ;
Ecrire (<var1>, <var2>, …) ;
Ecrire ("message") ;

I.3.9 Les structures de contrôle :

L'instruction conditionnelle (alternative):

a) si … alors…finsi

si (<condition>) **alors** <groupe d'instructions > ;
fsi ;

b) si … alors…sinon … finsi

si (<condition>) **alors** < groupe d'instructions1 > ;
sinon < groupe d'instructions2 >;
fsi ;

c) L'instruction cas …fincas ;

L'instruction cas permet de faire un choix parmi plusieurs possibilités suivant la valeur d'une expression qui doit être du type ordinal (entier, caractère ou booléen).

La première syntaxe de cette instruction est la suivante :

Cas <expression> **Vaut**
Domaine 1 : <bloc d'actions 1>
…
Domaine n : <bloc d'actions n>
Fincas ;

Remarque : Domaine peut être une liste de constantes.

La deuxième syntaxe de cette instruction est la suivante :

Cas<expression>**Vaut**
Domaine1 : <bloc d'actions 1>
…
Domaine n : <bloc d'actions n>
Sinon <bloc d'actions p>
Fincas ;

Dans ce cas si aucun élément d'aucun domaine n'est égal à <expression> alors le bloc d'actions p sera exécuté.

Les instructions itératives:

a) L'instruction Tantque … faire

Tant que (<conditions>)
Faire
<Groupe d'instructions >
Fait ;

b) L'instruction Pour

Pour <nom de variable>←<valeur initiale> **à** <valeur finale>
Faire
<Groupe d'instructions >
Fait ;

Pour <nom de variable>←<valeur initiale>**à**<valeur finale> **pas** <valeur>
Faire
<Groupe d'instructions >
Fait ;

c) L'instruction Répéter … jusqu'à

Répéter
<Groupe d'instructions>
Jusqu'à (<conditions>) ;

I.4 Les Actions Paramétrées : les procédures et les fonctions

Définitions *:*

a. Une variable locale est une variable qui ne peut être utilisée que dans l'action paramétrée ou le bloc où elle est définie. Par défaut, les variables locales sont temporaires. A la sortie de l'action paramétrée, les variables locales sont donc perdues.

b. Une variable globale peut être utilisée dans tout le programme.

Remarque : Les variables locales n'ont en particulier aucun lien avec des variables globales de même nom.

c. Paramètre formel, on parle aussi d'argument muet. Il s'agit de la variable utilisée dans le corps de l'action paramétrée.

d. Paramètre réel ou réel. Il s'agit de la variable (ou valeur) fournie lors de l'appel de l'action paramétrée. Certains langages, utilisent le terme *paramètre* pour paramètre formel et *argument* pour paramètre effectif.

Syntaxe d'une procédure:

Procédure Proc (<paramètres formels>)
Déclaration de variables locales
Début
<groupe d'instructions>
Fin ;

Syntaxe d'une fonction:

Fonction Fct (<paramètres formels>) : <Type du résultat>
Déclaration de variables locales
Début
<groupe d'instructions> ;
retourner (<résultat>) ;
Fin ;

<Type du résultat> désigne le type de la fonction, c'est-à-dire le type de la valeur qu'elle retourne qui peut être de n'importe quel type simple ou structuré.

Le corps de la fonction débute éventuellement par des déclarations de variables, qui sont locales à cette fonction. Il se termine par l'*instruction de retour à la fonction appelante*, retourner (return en C), dont la syntaxe est :

Retourner (*expression*);

La valeur de *expression* est la valeur que retourne la fonction. Son type doit être le même que celui qui a été spécifié dans l'en-tête de la fonction.

Passage des paramètres par adresse (entrée/sortie) ou par valeur (entrée)

Pour les distinguer on écrit E/ pour un paramètre en entrée et E/S pour un paramètre en sortie ou en entrée/sortie E/S.

Lorsqu'une variable est passée en argument à une action, on peut vouloir :

- Soit n'utiliser que la **valeur** de la variable passée en argument
- Soit utiliser la variable elle même dans le but de la modifier.

Dans le premier cas, on utilise le **passage** de paramètres **par valeur**. Dans le deuxième cas, on utilise le **passage** de paramètres **par adresse**.

Les paramètres par valeur d'une action sont traités de la même manière que les variables locales : lors de l'appel de l'action, les paramètres effectifs sont copiés dans l'espace réservé aux paramètres formels. L'action paramétrée travaille alors uniquement sur cette copie. Cette copie disparaît lors du retour au programme appelant. Cela implique en particulier que, si l'action modifie la valeur d'un de ses paramètres, seule la copie sera modifiée ; la variable du programme appelant, elle, ne sera pas modifiée.

Pour qu'une action modifie la valeur d'un de ses arguments, il faut qu'elle ait pour paramètre l'adresse de cet argument et non sa valeur. Par exemple, pour échanger les valeurs de deux variables, il faut écrire :

*Procedure Permuter (*E/S *A : entier,* E/S *B : entier)*
variable X : entier;
debut
    X←A;
    A← B;
    B←X;
fin;

Syntaxe de l'appel à une action :

L'appel à une procédure :

<nom de la procédure> (paramètres effectifs) ;

Exemple :
Algorithme traitement_procedure ;
Variable A, B : entier ;
*Procedure Permuter (*E/S *A : entier,* E/S *B : entier)*
variable X : entier;
debut
    X←A;
    A← B;
    B←X;
Fin ;
Début
    Lire (A, B);
    Permuter (A, B);
    Écrire (A, B) ;
Fin.

L'appel à une fonction :

<nom de variable> ← <nom de la fonction> (paramètres effectifs) ;

Exemple

```
Algorithme  traitement_fonction ;
      Variable  nb1, nb2, max : entier ;
     Fonction maximum (E/ n1 : entier, E/n2 : entier) : entier
      Début
          Si (n1>n2) alors retourner (n1)
                       sinon retourner (n2);
          Finsi ;
      Fin;

Début
   Lire (nb1, nb2);
  max←maximum (nb1, nb2);  ou bien écrire (maximum(nb1, nb2)) ;
  Écrire (max) ;
Fin.
```

Traduction des exemples en langage C :

```
#include <stdio.h>

void Permuter (int* A, int* B)
    {    int X ;
         X=*A ;
         *A=*B ;
         *B=X ;
    }

void main()
{  int A, B;
   scanf ("%d%d", &A, &B);
   Permuter (&A,&B);
   printf ("A=%d \n B=%d", A, B);
}
```

```
#include <stdio.h>

int maximum (int n1,  int n2 )
{
    if (n1>n2) return (n1);
          else  return (n2);
}

void main()
{  int nb1 nb2, max;
   scanf ("%d%d",&nb1, &nb2);
  max=maximum (nb1, nb2);
   printf ("La plus grande valeur est
              %d", max) ;
  // ou printf ("La plus grande valeur
     est %d", maximum(nb1, nb2)) ;
 }
```