Cours “Algorithmique et Structure des Données”

Licence L2 Informatique, semestre S3

Année 2019-2022

Problème → algorithme

Un problème à résoudre / à traiter doit d'abord être énoncé sous la forme d'un **cahier des charges** ; ensuite, il doit être :

- reformulé,
- précisé,
- spécifié,
- décomposé.

Algorithme

=

expression *dans un langage formel* d'une méthode de résolution par décomposition du traitement à effectuer en actions élémentaires, en précisant l'ordre de leur exécution.

Les données

Un schéma directeur de base :

Exemple : Calcul de l'aire d'un disque dont on connaît le rayon.

Les **données** sont les informations nécessaires au traitement ; un même traitement sera exécuté plusieurs fois, sur des *jeux de données* différents.
Les données doivent être **structurées** en fonction

- de la nature des informations qu'elles véhiculent,
- des codages acceptables par la machine,
- de la nature des traitements à effectuer.

N.B. Le terme "donnée" est générique, un résultat est une donnée au sens de l'algorithmique.

La structure des données

Les données peuvent être de différentes natures, suivant qu'elles doivent représenter des nombres, des lettres, des textes, etc. . . Une donnée peut être unique ou faire partie d'un ensemble (ex. un tableau de données).

Les données sont représentées par des variables.
Les variables sont d'un certain type.

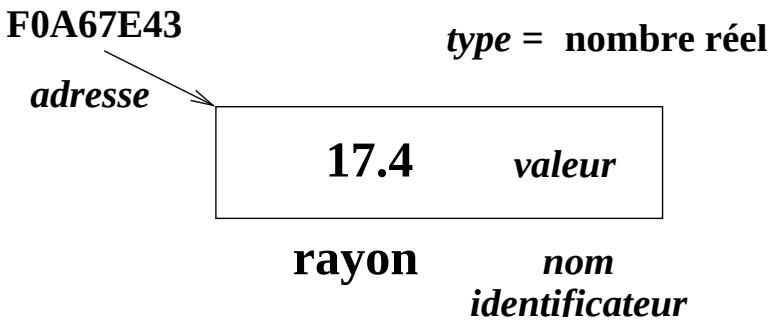
Cette structuration sera utile à la fois à :

- l'homme : les manipulations autorisées sur une variable dépendent de son type,
- la machine : la place mémoire qu'il faut allouer à une variable et les instructions valides qui la concernent dépendent de son type.

La notion de variable

Les **variables** servent à représenter les données, les résultats, etc...

Une **variable** correspond à :



- *(pour l'homme)* un objet, une quantité
  - qui reçoit un **nom** ou **identificateur**,
  - qui est d'un certain **type** (selon la nature de l'information),
  - qui peut prendre des **valeurs** ;
- *(pour la machine)* un emplacement mémoire
  - caractérisé par une **adresse** (lieu où la variable est stockée)
  - et une taille (place utilisée) qui dépend du *type* de la variable.

Programme = algorithme + structure de données

Adopter une bonne démarche algorithmique :

1. Affiner le cahier des charges :
   apporter des précisions, faire des choix, délimiter le domaine de validité
2. Analyser le sujet, le problème :
   imaginer une méthode de résolution, apporter des connaissances extérieures
3. Décrire l'**algorithme** et préciser la **structure des données**

---

4. Ecrire le **programme** dans un langage de programmation
5. Tester ce programme sur un jeu de données significatif

Résultat = **programme**

que l'on peut utiliser pour produire des résultats qu'il convient ensuite d'analyser.

Exercice :

CAHIER DES CHARGES :

Saisir une date tapée au clavier par l'utilisateur et lui dire si elle est valide.

CORRIGE : ..... À SUIVRE .....

Exemple d'algorithme

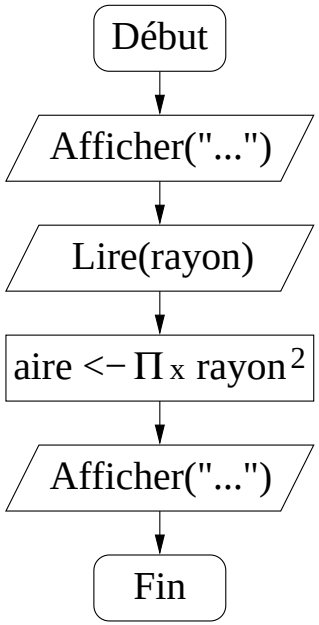
Cahier des charges : Calcul de l'aire d'un disque dont on connaît le rayon.

structure des données :

la *constante* π ; les *variables* rayon, aire : de type nombre réel ;

algorithme :

Organigramme



Pseudo-langage

Début
Afficher(“Longueur du rayon (en cm) ?”) ;
Lire(rayon) ;
aire $\leftarrow$ $\pi \times$ **rayon**^{2} **;**
Afficher(“L'aire mesure”, aire, “cm^2**”) ;**
Fin.

Pourquoi un “pseudo-langage” ?

L'algorithmique s'intéresse à la structuration d'une méthode pour résoudre un problème, indépendamment des particularités de tel ou tel langage.

Cela permet de :

- se focaliser sur la structure logique de la méthode à mettre en œuvre
- s'abstraire provisoirement des problèmes de syntaxe nécessairement liés à un langage de programmation
- décrire des méthodes qui pourront ensuite être programmées dans divers langages

Passage de l'algorithme à l'écriture du programme

Programme en **langage Pascal**, pour le calcul de l'aire d'un disque.

Structure des données

constante π ;

nombres réels rayon, aire

Algorithme

Début

Afficher("Longueur du rayon (en cm) ?") ;

Lire(rayon) ;

aire $\leftarrow$ $\pi \times$ rayon^2 ;

Afficher("L'aire mesure", aire, "cm^2") ;

Fin.

Programme

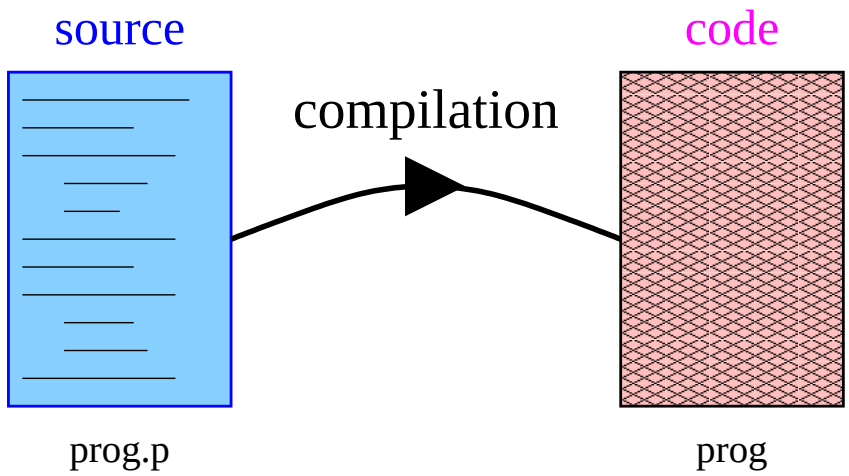
```
int main(){

float pi = 3.1416 ;
float rayon, aire ;

printf("Longueur du rayon (en cm) ? =") ;
scanf("%f",&rayon) ;
aire = pi * rayon * rayon ;
printf("L"aire mesure %f cm carrés",
aire) ;
}
```

Passage du programme écrit au programme exécutable

Comment passer de l'**écriture** du programme à son **exécution** ?
La plupart des langages utilisent un **compilateur**.



Principales fonctions du compilateur :

- contrôle de la syntaxe
- traduction en langage machine

Comment s'y prendre ?

Il n'y a pas de méthode systématique, mais plutôt une démarche :

- avoir une certaine intuition $\Leftarrow$ s'acquiert par l'expérience,
- être méthodique et rigoureux,
- se mettre mentalement à la place de la machine.

Notons par ailleurs que : il n'y a PAS de solution unique en algorithmique (de même qu'il n'y a pas de recette unique pour la tarte aux pommes !), mais il peut y avoir des algorithmes plus ou moins efficaces.

Composants de base

De même que des millions d'ADN différents résultent des combinaisons variées de 4 éléments invariables (ACGT), des millions d'algorithmes variés se fondent sur 4 familles d'instructions :

- affectation de variable,
- lecture / écriture,
- test $\Rightarrow$ instructions conditionnelles,
- boucles [i.e. répétitions ou itérations]

et ceci se retrouve dans toutes les familles de langage de programmation. (pour plus de précisions, voir le document annexe **FamLang.pdf**)

Déclaration d'une variable

Lorsqu'on choisit de représenter une donnée par une variable, on va **l'annoncer** en déclarant cette variable. On définit alors :

- son nom ⇒ le choisir court mais significatif du rôle que jouera la variable dans l'algorithme,
- son type ⇒ il sera choisi en fonction de la nature des valeurs que recevra cette variable et des manipulations qu'elle devra subir.

Exemples :
une variable nommée **dim**, de type entier, si elle représente la dimension d'un vecteur de données d'entrée ;
une variable nommée **phrase**, de type chaîne de caractères si elle contient un texte dont on veut compter le nombre de 'b'.

Affectation d'une variable

Déclarer une variable $\neq$ lui attribuer une **valeur**

Deux manières d'attribuer une valeur à une variable :

- en effectuant une **lecture**
  la valeur vient alors de l'extérieur ; elle est communiquée par l'utilisateur
- par une instruction d'**affectation**
  la variable (écrite à gauche du symbole d'affectation) reçoit pour valeur le résultat de l'évaluation (ex. calcul) du membre de droite

Exemple :

`x` $\leftarrow$ `3*a+2*(b-7)` que vaut x lorsque a vaut 4 et b vaut 10 ?

N.B. Le symbole $\leftarrow$ du pseudo-langage montre bien **le sens** de l'affectation, contrairement au signe ambigu $=$ utilisé en langage C.

Affectation de chaînes de caractères

Les chaînes de caractères sont toujours notées entre guillemets.

WARNING : Ne pas confondre le nom d'une variable avec son contenu, en particulier dans le cas des chaînes de caractères.

Pour éclairer ce point, comparons les deux algorithmes suivants :

Exemple n°1 :

```
Début
variable chaîne Riri, Fifi
Riri ← "Loulou"
Fifi ← "Riri"
Fin
```

Exemple n°2 :

```
Début
variable chaîne Riri, Fifi
Riri ← "Loulou"
Fifi ← Riri
Fin
```

Lecture

Une variable peut aussi recevoir une valeur de l'extérieur, en la demandant à l'utilisateur. Pour lire une valeur pour la variable **dim**, on écrira tout simplement, en pseudo-langage (c'est plus compliqué en C. . .) :

`Lire` **dim**

L'exécution du progamme s'arrête alors jusqu'à ce que l'utilisateur ait tapé une valeur, puis la touche "Entrée".

WARNING : si la valeur tapée n'est pas compatible avec le type de variable alors l'exécution sera interrompue avec un message d'erreur.
$\Rightarrow$ il est conseillé de poser une question à l'utilisateur pour l'informer de ce qu'on attend de lui.

Écriture

Pour écrire du texte à l'écran, on utilisera le mot de pseudo-langage Ecrire.
L'argument pourra être :

- une chaîne de caractères (entre guillemets, donc)
- le nom d'une variable (sa valeur sera alors affichée)
- une expression (le résultat de son évaluation sera affiché)
- une concaténation de plusieurs entités des genres ci-dessus, séparées par des signes +

Exemple :

```
Ecrire ‘‘Fifi’’ + Fifi + x + ‘‘ ’’ + 3*a-b
```

(penser à inclure des espaces - entre guillemets - pour rendre le tout lisible)

Tests - Valeurs et variables booléennes

Une condition portant sur des variables peut être, selon les valeurs des variables, soit vérifiée (*vraie*), soit non vérifiée (*fausse*). Le résultat de son évaluation est une valeur booléenne qui ne peut être que VRAI ou FAUX.

Plus généralement une variable de type booleen (sans accent, si écrit en pseudo-langage) ne peut prendre **que deux valeurs possibles**, VRAI ou FAUX.

Exemple :
La condition `(x<3) ET ((y NON= (x-1)) OU (y>2))`
est évaluée à VRAI quand x vaut 1 et y vaut 7 ;
elle est évaluée à FAUX quand x vaut 2 et y vaut 1.

Numériquement (en C, par exemple), une condition vérifiée est assimilée à la valeur 1, une condition fausse à la valeur 0. Inversement : zéro correspond à FAUX tandis que toute autre valeur entière (dont la valeur 1) correspond à VRAI.

Instructions conditionnelles

Parmi les nombreuses instructions qui constituent un programme, on peut souhaiter que certaines ne soient exécutées que sous certaines conditions, et omises si ces conditions ne sont pas remplies.

```
Si      (condition)
          instruction ;
FinSi
```

ou bien :

```
Si      (condition)
          instruction 1 ;
           ...
          instruction N ;
FinSi
```

⇐ *bloc d'instructions*

Dans tous les cas, l'exécution se continuera ensuite par les instructions qui suivent cette portion de programme "optionnelle".

Alternatives

On peut aussi souhaiter que telle instruction (ou bloc d'instructions) soit exécutée lorsque la condition est remplie (condition "VRAIE") et telle autre instruction ou bloc lorsqu'elle ne l'est pas (condition "FAUSSE").

```
Si      (condition)
          instruction A ;
Sinon
          instruction B ;
FinSi
```

ou bien :

```
Si      (condition)
          instruction A1 ;
           ...
          instruction Am ;
Sinon
          instruction B1 ;
           ...
          instruction Bp ;
FinSi
```

Choix multiples

```
Choisir       (expression)
        cas constante_1 :
              instruction 1 [ou bloc 1] ;
        cas constante_2 :
              instruction 2 [ou bloc 2] ;
               ...
        cas constante_N :
              instruction N [ou bloc N] ;
        ParDéfaut :
              instruction N+1 [ou bloc N+1] ;
FinChoisir
```

Si l'expression de contrôle a pour valeur constante_K alors toutes les instructions qui suivent celles étiquetées par "cas constante_K :" sont exécutées (de K à N inclus). Si l'expression n'a aucune des valeurs prévues dans la liste, alors seule l'instruction qui suit "ParDéfaut" est exécutée.

Aiguillages

Si l'on veut qu'une seule des instructions (ou blocs) soit exécutée, celle qui correspond à la valeur constante_K mais pas toutes celles qui suivent, il faut ajouter "break ;" à la fin de chaque suite d'instructions suivant un cas.

```
Choisir        (expression)
        cas constante_1 :
                instruction 1 [ou bloc 1] ;
                break ;
        cas constante_2 :
                instruction 2 [ou bloc 2] ;
                break ;
                 ...
        cas constante_N :
                instruction N [ou bloc N] ;
                break ;
        ParDéfaut :
                instruction N+1 [ou bloc N+1] ;
                break ;
FinChoisir
```

Répétitions - Itérations

On programme une répétition si l'on veut que la même instruction (contenant éventuellement une ou plusieurs variables) soit exécutée plusieurs fois.

Exemples d'exécution :

On répète cette ligne.	Ceci est la ligne 1.
On répète cette ligne.	Ceci est la ligne 2.
On répète cette ligne.	Ceci est la ligne 3.

On programme une itération si l'on doit répéter une action élémentaire pour construire un résultat *pas à pas*.

Exemple :

Calculer la somme d'une suite de nombres tapés par l'utilisateur.

Boucles **tant que**

```
TantQue     (condition)
            instruction ;
FinTantQue
```

ou bien :

```
TantQue     (condition)
              instruction 1 ;
               ...
              instruction N ;
FinTantQue
```

La condition est évaluée, puis l'instruction (le bloc) est / sont exécuté(s), seulement si la condition est VRAIE ; puis on remonte à l'évaluation de la condition, et ainsi de suite, *tant que* la condition reste VRAIE. Dès qu'elle devient FAUSSE, on passe aux instructions qui suivent la **boucle**.

Boucles **faire**

```
Faire
        instruction ;
TantQue (condition) ;
```

ou bien :

```
Faire
        instruction 1 ;
        ...
        instruction N ;
TantQue (condition) ;
```

Les instructions sont exécutées, puis la condition est évaluée : si elle est VRAIE alors on remonte en haut de la **boucle** pour exécuter une seconde fois les instructions, et ainsi de suite, *tant que* la condition reste VRAIE. Dès qu'elle devient FAUSSE, on passe à la suite du programme.

Boucles **pour**

Ce type de **boucle** ne s'utilise que lorsque *le nombre de passages* dans la boucle peut être contrôlé par une variable que l'on appellera compteur.

Pour (initialisation ; test_d'arrêt ; incrémentation)
instruction [ou bloc, entre Début et Fin] ;

L'**initialisation** fixe la valeur initiale du compteur (affectation de variable), le **test d'arrêt** porte sur la valeur du compteur, l'**incrémentation** est une expression qui définit le pas de variation du compteur d'un passage dans la boucle au suivant.

Exemple :

```
Pour (i←1; i<=10; i←i+1)
    Ecrire("Ceci est la ligne " + i);
FinPour
```