Algorithmique

Dominique Archambault
Master Technologie et Handicap — Université Paris 8

1 Introduction

1.1 Définition

Un algorithme
C'est une suite d'instructions qui, appliquée à un ensemble de données, conduit à un résultat.

- Si l'algorithme est correct, on obtiendra le résultat voulu.
- S'il est faux, le résultat est aléatoire.

De nombreux outils formels et théories ont été développés pour les décrire, les étudier, les comparer entre eux, etc.

C'est l'objet de l'**Algorithmique**

1.2 Conventions

Un langage non formel
- Indépendant des langages informatiques
  - ... pour se libérer du formalisme pour se concentrer sur la solution du problème
  - ... pour ensuite pouvoir implémenter facilement l'algorithme dans n'importe quel langage de programmation
- Compréhensible pour qui va le lire
- Un langage non formel MAIS **très rigoureux**

```
si x % 2 = 0 alors
    ecrire "x est pair"
sinon
    ecrire "x est impair"
finsi
```

2 Structures

Deux classes de concepts

Structures de données
- Constantes
- Variables
- Tableaux
- *Structures récursives (listes, piles, arbres, etc)*

Structures de contrôle
- Séquence
- Condition
- Itération

2.1 Structures de données

Variables

A quoi ça sert ?
Stocker provisoirement des données.

Qu'est ce qu'elle peut représenter ?
Des caractères, des nombres entiers ou réels, des booléens (vrai/faux)

Comment ?
Dans une zone de mémoire de l'ordinateur.

Une variable est une "boîte" dans laquelle on va stocker **une** valeur, correspondant à une zone de mémoire.

- elle a **identificateur** ou nom de variable
- à chaque instant elle a une valeur **unique**
- elle a un type, permettant de décoder le code en mémoire

Types de variable
C'est ce qui permet à la machine de savoir comment coder et décoder le contenu binaire d'une variable.

- caractères 'a', '0', ' ?'
- nombres entiers
  - sur un octet (de 0 à 255)
  - sur 2 octets (de 0 à 65 535 ou de -32 768 à +32 767)
  - sur 4 octets (de 0 à 4 294 967 296 ou de -2 147 483 648 à 2 147 483 647)
- réels
  - simple précision (de 10^{-45} à 10^{38})
  - double précision (de 10^{-324} à 10^{308})
- booléens (vrai/faux)

Déclaration
Déclarer une variable en précisant son type

```
var a : entier
var lettre : car
var pi : reel
```

Ce qui se passe :
- *Le système réserve Une zone de mémoire pour cette variable.*
- ***Et** il lui assigne un type, i.e. un codage permettant de savoir comment la coder et la décoder.*

Affectation
Assigner une valeur à une variable

```
a ← 0
lettre ← 'A'
pi ← 3.14159265
```

Ce qui se passe :
- *La valeur à placer dans la variable est codée selon son type, sous forme binaire.*
- *Ce code binaire est placé dans la zone mémoire correspondant à cette variable.*

2 règles importantes
- une variable contient toujours une valeur **unique**
- une variable doit toujours être initialisée

Expressions
On peut affecter à une variable le résultat d'un calcul.

```
var a, b, c : entier
var r : reel
a ← 2
b ← 2 + 3
    c ← b / a
    a ← 3 * b + c
    c ← c + 1
    r ← b / c
```

L'expression est d'abord évaluée, puis le résultat est affecté à la variable.

Opérateurs

- *opérateurs "habituels" :* | + | - | * | / | % |
- *opérateurs de comparaison :* | < | > | ≤ | ≥ | = | ≠ |
- *décalages de bits :* | << | >> |
- *etc*

2.2 Lecture/Écriture

Écriture
Consiste à écrire sur un flux (fichier, socket, sortie standard etc...). Par défaut affiche sur l'écran. Par convention on utilisera **EOL** pour revenir à la ligne.

```
ecrire "Hello world!"
a ← 1
pi ← 3.14159265
ecrire a, EOL, "La valeur de pi est ", pi
```

Écran

```
Hello world!
1
La valeur de pi est 3.14159265
```

Lecture
Consiste à lire sur un flux (fichier, socket, entrée standard etc...). Par défaut lire sur le clavier du terminal.

```
var a : entier
ecrire "Saisir un entier, SVP "
lire a
ecrire a+1
```

Écran

```
Saisir un entier, SVP
2
3
```

Lecture avec prompt

```
var a : entier
lire "Saisir un entier, SVP : ", a
ecrire "le suivant est ", a + 1
```

Écran

```
Saisir un entier, SVP :  2
le suivant est 3
```

Écriture : Machine $\Longrightarrow$ Environnement
Lecture : Machine $\Longleftarrow$ Environnement

2.3 Structures de contrôle

Séquence

Exécution ordonnée d'une suite d'instructions

```
debut
instruction1
instruction2
instruction3
fin
```

*Les instructions sont effectuées les unes après les autres de façon inconditionnelle, en **"séquence"**.*

Exemple :

```
var a : entier
debut
a ← 0
ecrire a
a ← 2
ecrire a
fin
```

Écran

```
0
2
```

Condition

Choix d'une séquence d'instructions ou d'une autre en fonction du résultat de l'évaluation d'une expression booléenne

```
si expression alors
  séquence d'instructions 1
sinon
  séquence d'instructions 2
finsi
```

*En fonction de l'évaluation de "`expression`", qui est de type booléen (valeurs possibles : Vrai ou Faux), seulement l'une des 2 séquences d'instruction sera réalisée. L'exécution se poursuit dans les 2 cas après le "**`finsi`**".*

Exemple :

```
var a : entier
debut
a ← 1
si a ≥ 0 alors
  ecrire a,"est positif"
sinon
  ecrire a,"est négatif"
finsi
fin
```

Écran

```
1 est positif
```

Variante

```
si expression alors
  séquence d'instructions
finsi
```

*Le "**`sinon`**" est facultatif.*

Itération

Répeter une séquence tant qu'une condition est réalisée

```
tantque expression faire
  séquence d'instructions
fait
```

Attention la séquence d'instruction doit modifier au moins une des variables présentes dans l'expression booléenne.

Exemple :

```
var i : entier
debut
i ← 0
tantque i < 3 faire
    i ← i+1
    ecrire i
fait
ecrire "terminé"
fin
```

Écran

```
1
2
3
terminé
```

Variante

```
pour v allantde val1 a val2 faire
  séquence d'instructions
fait
```

Permet d'être sûr qu'on exécute la séquence d'instructions un nombre précis de fois.

Code équivalent

```
v ← val1
tantque v ≤ val2 faire
  séquence d'instructions
  v ← v + 1
fait
```

Variante

```
repeter
  séquence d'instructions
tantque expression
```

Permet d'être sûr qu'on exécute la séquence d'instructions au moins une fois.

Code équivalent

```
séquence d'instructions
tantque expression faire
  séquence d'instructions
fait
```

2.4 Un exemple : Algorithme d'Euclide

Il s'agit de calculer le PGCD de 2 nombres.

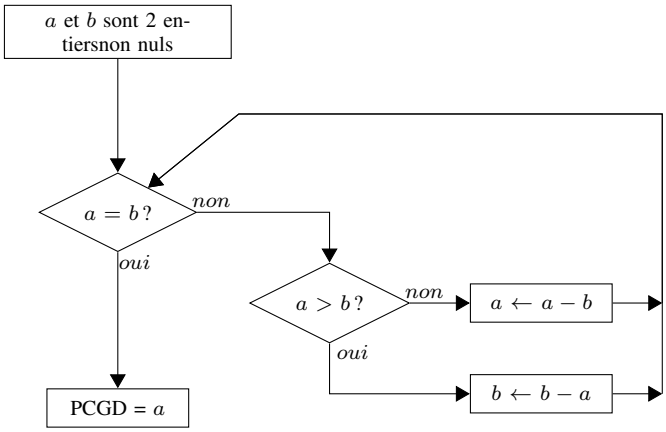
calcul du PGCD de a et b

- si $a = b$, le pgcd est a
- si $a < b$, le pgcd est le pgcd de a et $b - a$
- si $b < a$, le pgcd est le pgcd de b et $a - b$

et on recommence jusqu'à ce que a et b soient egaux...

Algorithme d'Euclide

Représentation graphique




```
var a, b  : entier
debut
lire "saisir a et b", a, b
tantque a ≠ b faire
  si (a > b) alors
    a ← a-b
  sinon
    b ← b-a
  finsi
fait
ecrire "Le PGCD est ", b
```

Implémentation en C

```
int a, b;
// lecture de a et b
while (a != b) {
    if (a>b) a=a-b;
    else b=b-a;
}
printf("Le PGCD est %d.", a);
```

Implémentation en Python

```
// lecture de a et b
while a!=b:
   if a>b:
      a = a - b
   else:
      b = b - a
print("Le␣PGCD␣est", b)
```

Implémentation en Pascal

```
program Euclide(input, output);
var a, b, r: integer;
begin
writeln('Saisir␣a,␣b');
readln(a); readln(b);
while a <> b do begin
    if a>b then
        a := a - b
    else
        b := b - a
    end;
writeln('Le␣PGCD␣est', b);
end.
```

Implémentation en java

```
class Euclide {
  public static void main(String args[]) {
    int a, int b;
    // initialisation de a et b
    while (a != b) {
      if (a>b) a=a-b;
      else b=b-a;
    }
    System.out.println("Le␣PGCD␣est␣"+b);
  }
}
```

Implémentation en Shell

```
a=$1; b=$2; r=1
until [ $a -eq $b ]; do
  if [ $a -gt $b ]; then
    a=$(($a - $b))
  else
    b=$(($b - $a))
  fi
done
echo "PGCD($1,␣$2)=␣$a"
```

Implémentation en Prolog

```
pgcd(X,X,X).
pgcd(X,Y,D) :- X < Y, Z is Y-X, pgcd(X,Z,D).
pgcd(X,Y,D) :- Y < X, pgcd(Y,X,D).
```

Implémentation en Lisp

```
(defun pgcd (a b)
  (cond
    ((= a b) a)
    ((< a b) (pgcd a (- b a)))
    (t       (pgcd b (- a b))) ))
```

3 Notions avancées

3.1 Tableaux

Manipuler des ensembles de données.

- il serait fastidieux d'utiliser des variables pour des dizaines de données
- on ne connaît pas toujours le nombre de données à traiter

Déclaration

```
var tableau[20] : entiers
//déclaration d'un tableau de 20 entiers
```

Utilisation

```
tableau[0] ← 2;
i ← 3;
m ← m + tableau[i]
```

Exemple : calcul de moyenne

```
var note[20], s, i : entier
var m : reel
debut
ecrire "saisir␣les␣20␣notes"
i ← 0
tantque i < 20 faire
  lire "note", note[i]
  i ← i + 1
fait
s ← 0
i ← 0
tantque i < 20 faire
  s ← s + note[i]
  i ← i + 1
fait
m ← s / 20.0
ecrire "la␣moyenne␣est␣de␣", m
fin
```

A savoir

- Si un tableau est de taille n, ses éléments sont numérotés de 0 à $n-1$. Le premier est donc `t[0]`.
- On ne peut pas accéder à un élément en dehors du tableau. Si t à 10 éléments, faire référence à t[10] ou t[12] sont des erreurs.

Tableaux à plusieurs dimensions
Ça ne pose pas de problème. On les déclarera ainsi :

```
var tableau[10,5] : entier
```

On ne sait pas toujours la taille du tableau au moment où l'on écrit le programme ! Les tableaux dynamiques permettent de déclarer la taille du tableau lors de l'exécution.

Déclaration de tableaux dynamiques
On indique pas la taille, tout simplement

```
var tab[] : entier
```

Allocation de tableaux dynamiques
On indique la taille du tableau dans le cours du programme

```
dimension tab[10]
```

Exemple : calcul de moyenne

```
var note[], i, s, n : entier
var m : reel
debut
lire "Combien␣de␣notes␣à␣saisir?␣", n
dimension note[n]
// suite identique à l'exemple plus haut
// ...
```

3.2 Décomposer un problème : les fonctions

Des gros algorithmes
Si un problème est complexe, l'algorithme sera long.

Longs algorithmes
Il est très difficile de travailler sur un algorithme long.
⇒ Une bonne solution consiste à diviser le problème est sous problèmes.

Traitements répétés
On a souvent les mêmes traitements à faire sur des données à différents endroits d'un algorithme.
⇒ Grouper ces traitements à un seul endroit et l'appeler à chaque fois que c'est nécessaire.

Un premier exemple

```
fonction PGCD
                (var a, b : entier par valeur)
var r : entier
debut
    si b > a alors
        r ← a
        a ← b
        b ← r
    finsi
    r ← a % b
    tantque r ≠ 0 faire
        a ← b
        b ← r
        r ← a % b
    fait
    retourner b
fin
```

Appel de fonction

```
fonction PGCD
              (var a, b : entier par valeur)
var r : entier
debut
    ...
    retourner b
fin
```

```
debut
ecrire PGCD(15, 24)
ecrire PGCD(21, 28)
fin
```

Arguments de fonctions

Portée de variable
On appelle portée d'une variable la période pendant laquelle cette variable existe.

Passage par valeur
La valeur de la variable est passée à la fonction. Elle est recopiée dans une variable locale de la fonction.

```
fonction uneFonction
                  (var a : entier par valeur)
debut
    ...
fin
```

Exemple :

```
fonction carre
                  (var a : entier par valeur)
var b : entier
debut
    b -> a * a
    retourner b
fin
```

```
var a, b : entier
debut
a ← 2
b ← carre(a)
a ← carre(3)
fin
```

Passage par reference
C'est la référence de la variable (on parle aussi de son adresse) qui est passée à la fonction. Si elle est modifiée par la fonction c'est donc la variable passée à la fonction qui est modifiée.
Appellations équivalentes : passage par adresse, passage par variable

```
fonction uneFctn
                  (var a : entier par reference)
debut
    ...
fin
```

Exemple :

```
fonction auCarre
              (var a : entier par reference)
debut
    a -> a * a
    retourner a
fin
```

```
var a, b : entier
debut
a ← 2
b ← auCarre(a)
// a ← auCarre(3)   IMPOSSIBLE
// a ← auCarre(a+2)   IMPOSSIBLE
fin
```

Procedures
Dans certains cas une fonction n'a pas de valeur de retour. On appelle ces fonctions des procédures.

```
fonction auCarre
            (var a : entier par reference)
debut
    a ← a * a
fin
```

```
var a, b : entier
debut
a ← 2
auCarre(a)
b ← a
fin
```

3.3 Récursivité

Fonction récursive

Une fonction qui s'appelle elle-même
Exemple : calcul de factorielle.

$$n! = 1 \times 2 \times ... \times (n-1) \times n$$

On peut la calculer par récurrence : $n! = n \times (n-1)!$

```
fonction factorielle
                (var n : entier par valeur)
// On suppose n > 0
debut
    si n = 1 alors
        retourner 1
    sinon
        retourner n * factorielle(n-1)
    finsi
fin
```

Reprenons l'algorithme d'Euclide, qui est initialement formulé de façon récursive (voir plus haut).

```
fonction PGCD(var a, b : entier par valeur)
var r : entier
debut
    si a == b
      alors retourner a
      sinon
        si a < b
          alors retourner PGCD(a,b-a)
          sinon retourner PGCD(b,a-b)
        finsi
    finsi
fin
```

4 Notions complémentaires

Preuve de programme
- Correction
- Complétude
- Terminaison

Complexité
Evaluer le nombre d'instructions effectuées en fonction du nombre de données en entrée.