

## Attention !

Les transparents ne contiennent pas la totalité du cours : ils sont faits pour être complétés par ce qui est dit en cours, et ce qui est vu en TD/TP.

Connaître le contenu des transparents par coeur n'est pas une garantie de réussir le module.

## Algorithmique et programmation 1 - Cours 2

# Qu'est-ce qu'un algorithme ?

L1 M-I-SPI – Université de Lorraine  
Marie Duflot-Kremer  
avec l'aide des collègues de Nancy et Metz

Transparents disponibles sur la plateforme de cours en ligne

## Un algorithme ? Je connais déjà !

### Déjà vu des algorithmes

- en maths au lycée,
- peut-être pour votre plaisir.

### Dans ce cours

- (re-)poser les bases,
- aller plus loin,
- acquérir une méthode de résolution de problèmes
  - ... pour faire des algorithmes plus clairs/efficaces,
  - ... pour les rendre utilisables sur de plus gros problèmes.

## Démarche pour résoudre un problème

- Énoncer clairement le problème à résoudre  
~> **spécification**
- Trouver *avec les mains* un moyen de le résoudre  
~> **analyse**
- Écrire formellement cette solution en pseudo-code  
~> **algorithme**
- Traduire ce pseudo-code dans un langage de programmation  
~> **programme**

## Spécification

Avant de savoir comment faire, il faut savoir précisément ce qu'on veut faire :

- Quelles sont les données nécessaires ?
  - Combien d'objets ?
  - Quels objets ? (nombres entiers, à virgule, mots, tableaux,...)
- Quel est le résultat attendu ?
  - Quel objet (nombre entier, à virgule, mot, tableau,...) ?
  - Quel est son lien avec les données ?

## Identifier données et résultat

Calcul de la moyenne de trois notes

Données :

Résultat :

Tri des valeurs contenues dans un tableau

Données :

Résultat :

Trouver tous les mails envoyés par un de ses amis

Données :

Résultat :

Changement de la couleur de fond d'une image

Données :

Résultat :

## L'analyse

- Deuxième étape pour résoudre un problème
- Comprendre le problème et essayer de trouver une méthode de résolution
- Souvent le plus difficile à faire
- En général plusieurs solutions existent pour un même problème

Premier problème

Trouver la plus grande valeur parmi deux nombres entiers

## L'algorithme

“Algorithme = procédure de calcul bien définie qui prend en entrée une valeur ou un ensemble de valeurs et qui produit, en sortie, une valeur ou un ensemble de valeurs.” [CLR90]

- Décomposer le travail en instructions élémentaires
  - ↪ chaque instruction va devoir être exécutée par un ordinateur
- Formaliser les idées trouvées lors de l'analyse
  - ↪ besoin d'un langage algorithmique précis

Attention !

L'algorithme doit fonctionner quelles que soient les valeurs (acceptables) des données !

## Le programme

Dernière étape de résolution : traduire l'algorithme dans un langage de programmation, que l'ordinateur peut comprendre.

### En "vrai"

Le programme en langage dit "évolué" (Python, C, Java, C++, Caml, Pascal,...) va encore passer par une étape pour le rendre assez simple pour être exécuté. Mais ça c'est le travail de l'ordinateur.

Intérêt des langages de programmation évolués :

- Plus faciles à comprendre par l'utilisateur
- Assez formels pour être compilés/interprétés

↪ un intermédiaire entre le coeur de la machine et nous

**Pour le cours d'AP1 nous utiliserons le langage Python**

## De l'algorithme au programme

Les deux sont en général très similaires.

```
# Algorithme Moyenne
# Calcule la moyenne de trois
# notes de type entier
Variables
| a, b, c, som : entier
| moy : réel
Début
| a ← 3
| b ← 5
| c ← 2
| som ← a+b+c
| moy ← som/3
| afficher("la somme vaut ", som)
| afficher("et la moyenne ", moy)
Fin
```

```
# Programme moyenne.py
# Calcule la moyenne de
# trois notes de type int
# Variables
# a, b, c, som : int
# moy : float
a = 3
b = 5
c = 2
som = a+b+c
moy = som/3
print("la somme vaut ", som)
print("et la moyenne ", moy)
```

- L'algorithme est en français,
- et un peu plus proche de la formulation naturelle.

Il y a presque 2000 ans...

### La méthode de Héron d'Alexandrie ( $\simeq 50-100$ ap JC)[Hér98]

Puisque 720 n'a pas son côté rationnel, on peut obtenir son côté avec une très petite différence comme suit. Comme le premier nombre carré successeur de 720 est 729 qui a 27 pour côté, on divise 720 par 27. Cela donne  $26\frac{2}{3}$ . On ajoute 27, ce qui fait  $53\frac{2}{3}$  et l'on en prend la moitié, soit  $26\frac{1}{2}\frac{1}{3}$ . Le côté de 720 sera par conséquent très proche de  $26\frac{1}{2}\frac{1}{3}$ . En fait, si l'on multiplie  $26\frac{1}{2}\frac{1}{3}$  par lui-même, le produit est  $720\frac{1}{36}$ , de sorte que la différence sur le carré est  $\frac{1}{36}$ . Si l'on désire rendre la différence inférieure encore à  $\frac{1}{36}$ , on prendra  $720\frac{1}{36}$  au lieu de 729, et en procédant de la même façon, on trouvera que la différence résultante est beaucoup moindre que  $\frac{1}{36}$ .

Problèmes

Ce qui nous gêne dans cette méthode :

- difficile à comprendre
  - les données font partie de la méthode
- ↪ comment faire si on change les données ?
- formulation brouillonne
- ↪ besoin de formaliser/clarifier tout ça

## Variables : à quoi ça sert ?

- Un problème de la méthode précédente : on raisonne sur un exemple
- ~> Comment le généraliser ?
- Besoin de donner un nom
  - à la valeur de départ,
  - aux valeurs intermédiaires,
  - au résultat.
- ~> Besoin d'introduire des variables

## Variables : informatique vs mathématiques

En mathématiques	En informatique
inconnue <ul style="list-style-type: none"> <li>• à laquelle on va donner une valeur (fonction)</li> <li>• à déterminer (équation)</li> </ul>	nom d'une <u>case</u> en mémoire où stocker une information
En nombre infini	En nombre fini
"non réutilisables"	réutilisables

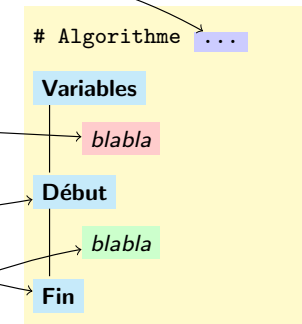
## Type des variables

Pourquoi donner un type aux variables ?

- Une place dans la mémoire, mais quelle place ?
  - ~> Besoin de savoir quelle taille mémoire réserver pour notre variable.
- Dans l'ordinateur, tout est stocké en binaire.
- Que code 01000111 ?
  - 
  - 
  - ~> Pour que l'ordinateur s'y retrouve, il doit savoir le type de ce qu'il stocke.
- Opérations définies pour des objets de type précis
  - ~> On ne peut pas additionner un mot et un réel

## Forme générale d'un algorithme

- Nom de l'algorithme,
- liste et type des variables utilisées dans l'algorithme,
- mots clefs : délimitent le corps de l'algorithme,
- instructions de l'algorithme, soigneusement indentées.



## Un langage algorithmique

Besoin d'un langage précis pour écrire les algorithmes.

**Pour le cours d'AP1 nous l'appellerons pseudo-code**

Peut faire plein de choses :

- définition des variables
- affectation
- commentaires
- saisie et affichage
- conditionnelles
- boucles
- ...

## Déclaration des variables

- À faire dans le préambule de l'algorithme.
- On donne son nom et le type de valeur qu'elle va recevoir

### Attention

- le nom est une suite de lettres, chiffres et "\_",
- il commence par une lettre
- la casse (différence MAJUSCULES/minuscules) compte
- si on fait une faute d'orthographe, ce n'est plus la même variable !!

## Déclaration des variables (2)

Types simples existants :

- entier naturel (0, 12, 5,...),
- entier (-6, 20, 1,...),
- réel (3.25,  $-1.36 \times 10^{12}$ ),
- caractère ('a', '%', '9', ...),
- chaîne de caractères ("toto", "2\*3=12", ...),
- booléen (vrai, faux)

```
# Algorithme Exvariables
Variables
| x : entier
| moyenne, t : réel
| message : chaîne de
|           caractères
| drapeau : booléen
Début
| ...
Fin
```

## Affectation

Affecter une valeur à une variable

= écrire dans la case mémoire correspondante  
= remplacer ancienne par nouvelle valeur.

- En pseudo-code, l'affectation se note  $\leftarrow$ ,
- on prend la valeur de ce qu'il y a à droite...
- ... et on la stocke dans la variable de gauche,
- $x \leftarrow 6$  signifie : la variable x reçoit la valeur 6,
- $x \leftarrow x+1$  signifie : la variable x reçoit la valeur de x plus 1.

## Affectation (2)

### Variable et type

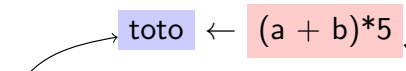
Une variable est définie avec un type, et donc :

- on l'initialise avec une valeur de ce type
- on change toujours une valeur pour une autre du même type

### Attention

- Par défaut une variable contient n'importe quoi  
 $\rightsquigarrow$  la dernière chose que l'ordinateur a écrit dans cet espace mémoire
- Il faut toujours initialiser une variable avant d'utiliser sa valeur

## Affectation (3)

- 
- À gauche : une variable
    - on évalue l'expression
    - on en déduit son type
    - on affecte la valeur à la variable si elles ont le même type
  - À droite : une expression
    - on évalue l'expression
    - on en déduit son type
    - on affecte la valeur à la variable si elles ont le même type
  - $\rightsquigarrow$  sinon, **erreur !**

### Un exemple

```

x   y
?   ?

x ← 12
y ← x/2 - 5
x ← y + x

```

## Commentaires

Un algorithme/programme est fait pour être lu par plusieurs :

- À la fac :
  - vous, pour le modifier, parfois après plusieurs jours/mois
  - les profs, pour vous aider/noter
  - d'autres étudiants lors de projets communs
- Au boulot :
  - vous, pour le modifier, parfois après plusieurs jours/mois
  - votre responsable, pour vérifier l'avancée/tester/corriger
  - vos collègues pour vérifier/modifier/compléter,
  - pourquoi pas un client/utilisateur

Il faut donc impérativement des explications

### Attention

Un programme sans commentaires sera fortement sanctionné

## Commentaires (2)

Qu'est-ce qu'un commentaire ?

- Une explication pour les humains qui lisent l'algorithme,
- n'a aucune influence sur le déroulement de l'algorithme,
- commence par # et va jusqu'à la fin de la ligne.

Quoi écrire dans un commentaire ?

- Au début de l'algorithme :
  - dire ce qu'on va faire,
- dans le corps de l'algorithme :
  - dire ce que font les blocs d'instructions importants,
  - expliquer une instruction qui n'est pas simple.

## Commentaires (3)

Exemple de commentaires utiles :

```
...
montant ← prix × reduc/100
                # calcule le montant de la réduction
prixreduit ← prix - montant
                # calcule le prix après réduction
...
```

Exemple de commentaires inutiles :

```
...
montant ← prix × reduc/100
                # montant reçoit le prix multiplié
                # par un centième de la réduction
prixreduit ← prix - montant
                # prixreduit reçoit prix moins le montant
...
```

L'utilité se verra mieux sur des algorithmes plus compliqués.

## Opérateurs

Dépendent du type de variable considéré :

- entiers : +, -, \*, // (division entière), mod (reste), comparaisons (=, <, ≤, ≥, >, ≠)
- réels : +, -, \*, /, comparaisons (=, <, ≤, ≥, >, ≠)
- booléens : et, ou, non, comparaisons (=, ≠)
- chaînes de caractères : longueur, + (concaténation), comparaisons
- ... et d'autres que l'on pourra introduire suivant les besoins

## Affichage

- Une fois le résultat calculé, on veut le voir
- ↪ on utilise l'instruction `afficher(x)`
- on peut aussi afficher la valeur d'une expression :  
`afficher((3*x)+2)`

### Remarque

Pour AP1 les données sont soit codées dans l'algorithme soit saisies et le résultat est affiché à l'écran. Dans la suite ce sera plus subtil (fonctions de calcul, de test,...)

## Saisie

Pour tester l'algorithme, besoin de choisir les valeurs de certaines variables :

- au départ
- parfois en cours d'algorithme

Deux solutions :

- les écrire dans le programme
  - `x ← 2`
    - ↪ ne fonctionne pas pour la changer en cours d'algorithme
- les demander à l'utilisateur
  - on attend que l'utilisateur tape une valeur,
  - puis on la met dans une variable (par exemple x)
  - ↪ on utilise l'instruction `x ← saisir()`
  - ↪ on peut préciser un message à afficher  
`x ← saisir ("Entrez un entier ")`

## Constantes symboliques

- Parfois besoin de stocker des valeurs qui ne changeront jamais
  - valeur approchée de  $\pi$ , du nombre d'or,...
  - une valeur importante qu'on veut écrire une fois et utiliser partout,
  - un message qui sera affiché plusieurs fois,
- on utilise alors des constantes,
  - définies en début de programme, avant les variables,
  - jamais modifiées,

Gérées différemment ou pas du tout selon les langages

## Vers un algorithme pour la méthode de Héron

### Spécification

- **Entrée :**
  - **Problème :** résoudre approximativement  $\sqrt{A}$  lorsque  $A$
  - **Résultat :** une approximation de  $\sqrt{A}$
- En fait ce n'est pas assez précis
  - Il manque la précision (mais Héron ne le dit pas)

## La méthode de Héron d'Alexandrie (2)

### La méthode de Héron d'Alexandrie ( $\simeq 50-100$ ap JC)[Hér98]

Puisque 720 n'a pas son côté rationnel, on peut obtenir son côté avec une très petite différence comme suit. Comme le premier nombre carré successeur de 720 est 729 qui a 27 pour côté, on divise 720 par 27. Cela donne  $26 \frac{2}{3}$ . On ajoute 27, ce qui fait  $53 \frac{2}{3}$  et l'on en prend la moitié, soit  $26 \frac{1}{2} \frac{1}{3}$ . Le côté de 720 sera par conséquent très proche de  $26 \frac{1}{2} \frac{1}{3}$ . En fait, si l'on multiplie  $26 \frac{1}{2} \frac{1}{3}$  par lui-même, le produit est  $720 \frac{1}{36}$ , de sorte que la différence sur le carré est  $\frac{1}{36}$ . Si l'on désire rendre la différence inférieure encore à  $\frac{1}{36}$ , on prendra  $720 \frac{1}{36}$  au lieu de 729, et en procédant de la même façon, on trouvera que la différence résultante est beaucoup moindre que  $\frac{1}{36}$ .

## La méthode de Héron d'Alexandrie (3)

### La méthode de Héron d'Alexandrie ( $\simeq 50-100$ ap JC)[Hér98]

Puisque 720 n'a pas son côté rationnel, on peut obtenir son côté avec une très petite différence comme suit. Comme le premier nombre carré successeur de 720 est 729 qui a 27 pour côté, on divise 720 par 27. Cela donne  $26 \frac{2}{3}$ . On ajoute 27, ce qui fait  $53 \frac{2}{3}$  et l'on en prend la moitié, soit  $26 \frac{1}{2} \frac{1}{3}$ . Le côté de 720 sera par conséquent très proche de  $26 \frac{1}{2} \frac{1}{3}$ . En fait, si l'on multiplie  $26 \frac{1}{2} \frac{1}{3}$  par lui-même, le produit est  $720 \frac{1}{36}$ , de sorte que la différence sur le carré est  $\frac{1}{36}$ . Si l'on désire rendre la différence inférieure encore à  $\frac{1}{36}$ , on prendra  $720 \frac{1}{36}$  au lieu de 729, et en procédant de la même façon, on trouvera que la différence résultante est beaucoup moindre que  $\frac{1}{36}$ .



## Vers un algorithme pour la méthode de Héron (2)

### Solution partielle :

- Trouver  $A_0$  le plus petit entier  $> A$  qui soit un carré
- Prendre  $a_0$  le “côté” (= la racine) de  $A_0$
- Répéter  $k$  fois : remplacer  $a_0$  par  $\frac{1}{2} \left[ \frac{A}{a_0} + a_0 \right]$
- on obtient la relation  $a_n = \frac{1}{2} \left( \frac{A}{a_{n-1}} + a_{n-1} \right)$

### Théorème

La limite  $\lim_{n \rightarrow +\infty} a_n$  existe et vaut  $\sqrt{A}$

- On s'arrête quand on est assez près de la solution ,  
 $\rightsquigarrow$  notion de *assez près* à clarifier dans la spécification,
- il nous manque des outils pour en faire un véritable algorithme,  
 $\rightsquigarrow$  à revoir dans le cours sur les boucles

## Vers un algorithme pour la méthode de Héron (3)

- variables  $a_n$ , avec  $n \rightarrow +\infty$
- $\rightsquigarrow$  un nombre infini de variables ?
- en maths, oui. En info, impossible.
- On va faire du “recyclage”,
- quand la nouvelle valeur est connue, on peut oublier les précédentes,
- on peut écrire  $a \leftarrow (A/a + a)/2$ ,
- ici une seule variable pour calculer le terme de la suite.

## Qu'est-ce qu'un bon algorithme ?

### Trois critères importants :

- correct  
 $\rightsquigarrow$  fait précisément ce qu'on lui demande, pour toutes les données acceptables
- efficace  
 $\rightsquigarrow$  entre deux algorithmes corrects on préfère le plus rapide
- compréhensible  
 $\rightsquigarrow$  doit pouvoir être compris/réutilisé/modifié par d'autres

## Écrire des algorithmes corrects

Ca marche sur un exemple. Oui, et alors ?

- $2 + 2 = 2 \times 2 = 4...$  et pourtant la somme n'est pas le produit
- un algorithme peut donner le bon résultat sur un exemple... et être complètement faux!!!

### Comment faire ?

- Pour de petits programmes, bien les tester
  - tester plusieurs valeurs,
  - tester les cas limites (0 ? une valeur maximale autorisée ?, ...),
  - s'il y a des tests dans l'algorithme, essayer les différents cas
- Pour des programmes critiques, d'autres méthodes
  - Preuve formelle
  - Model-checking
  - ...

Le test peut prouver l'existence d'une erreur, pas son absence

## Écrire des algorithmes efficaces

### Idée

- Plusieurs algorithmes corrects pour le même problème,
- on veut choisir le meilleur,
- on va comparer leur temps d'exécution
- (on peut aussi comparer l'espace mémoire utilisé)

### Temps d'exécution mesuré

- En termes de nombre d'opérations élémentaires (+, \*, ←, ...)
- défini comme une fonction de la taille des données

## Efficacité d'un algorithme - Exemple

```
# Algorithme FoisHuit
# Multiplie un entier par 8
# sans utiliser l'opérateur *
Variables
| a, res : entier
Début
| a ← saisir("Entrez un entier : ")
| res ← a + a
| res ← res + a
| res ← res + a
| res ← res + a
| res ← res + a
| res ← res + a
| res ← res + a
| afficher("Le produit : ", res)
Fin
```

```
# Algorithme FoisHuitmalin
# Multiplie un entier par 8
# sans utiliser l'opérateur *
Variables
| a, res : entier
Début
| a ← saisir("Entrez un entier : ")
| res ← a
| res ← res + res
| res ← res + res
| res ← res + res
| afficher("Le produit : ", res)
Fin
```

- FoisHuit :      additions et      affectations
- FoisHuitmalin :      additions et      affectations

## Histoire de comparer ...

Comment évolue le temps de calcul en fonction de la taille des données ?

$n \backslash$ complexité	$\log_2(n)$	$n$	$n \log_2(n)$	$n^2$	$n^3$	$2^n$
$10^1$	$3,3\mu s$	$10\mu s$	$33\mu s$	$0,1ms$	$1ms$	$1ms$
$10^2$	$6,6\mu s$	$0,1ms$	$0,66ms$	$10ms$	$1s$	$4.10^{16}a$
$10^3$	$9,9\mu s$	$1ms$	$9,9ms$	$1s$	$16,6m$	$\infty^*$
$10^4$	$13,3\mu s$	$10ms$	$0,13s$	$1,5m$	$11,5j$	$\infty$
$10^5$	$16,6\mu s$	$0,1s$	$1,64s$	$2,7h$	$31,7a$	$\infty$
$10^6$	$19,9\mu s$	$1s$	$19,9s$	$11,5j$	$3.10^4a$	$\infty$

\*  $\infty$  représente un temps  $> 10^{100}$  années

## Histoire de comparer ...(2)

Quelle taille de donnée peut-on traiter en un temps de calcul donné ?

temps \ complexité	$\log_2(n)$	$n$	$n \log_2(n)$	$n^2$	$n^3$	$2^n$
1s	$\infty^*$	$10^6$	$6,3.10^4$	$10^3$	$10^2$	19
1mn	$\infty$	$6.10^7$	$2,8.10^6$	$7.10^3$	$4.10^2$	25
1h	$\infty$	$36.10^8$	$1,3.10^8$	$6.10^4$	$15.10^2$	31
1j	$\infty$	$8,6.10^{10}$	$2,7.10^9$	$2,9.10^5$	$44.10^2$	36
1an	$\infty$	$3,2.10^{13}$	$8.10^{11}$	$5,7.10^6$	$3,2.10^4$	44

\*  $\infty$  représente une taille  $> 10^{100}$

## Efficacité - Complexité

Pas toujours possible de trouver la solution à un problème en un temps linéaire.

### Définition

La complexité en temps d'un problème est l'ordre de grandeur du temps que mettrait l'algorithme le plus efficace pour résoudre ce problème

### Exemples

- Tris : des algorithmes en  $n$  = nombre de valeurs à trier
- Recherche de motifs : des algorithmes en  $n$  = taille du texte,  $m$  = taille du mot
- Sac à dos : on ne sait pas faire mieux que  $n$  = taille des objets

## Ce que veulent les entreprises

“On ne recherche pas forcément une technologie mais une adaptativité, une méthodologie du travail, savoir pondre un algo, réfléchir.”

Cédric E. Chef de projet, CGI  
tuteur d'un stagiaire M2 de l'UL

- La démarche algorithmique est la base de tout apprentissage de la programmation, quel que soit le langage,
- c'est ce qu'on va apprendre dans le cours d'AP1.

## Écrire des algorithmes compréhensibles

- Nommer les variables de manière intelligente,
  - `somme` est plus parlant que `x`
- Indenter
  - décaler plus ou moins les lignes pour augmenter la lisibilité
  - important dans les algorithmes et la plupart des langages
  - **absolument nécessaire** dans Python
- Commenter !!!
- Mettre des affichages clairs à l'écran
  - `La somme : 8 et le produit : 12` plutôt que `8 12`



Th. Cormen, Ch. Leiserson, and R. Rivest, *Introduction à l'algorithmique*, Dunod, 1990.



Héron, *Metrica III,I,8, d'après : Caveing M., l'irrationalité dans les mathématiques grecques jusqu'à euclide*, p. 28, Septentrion, 1998.

## Sources