



Ada Lovelace



Alan Turing



Grace Hopper

## Information, Calcul et Communication

### Module 1 : Calcul

## Leçon I.2 : Qu'est-ce qu'un algorithme ?

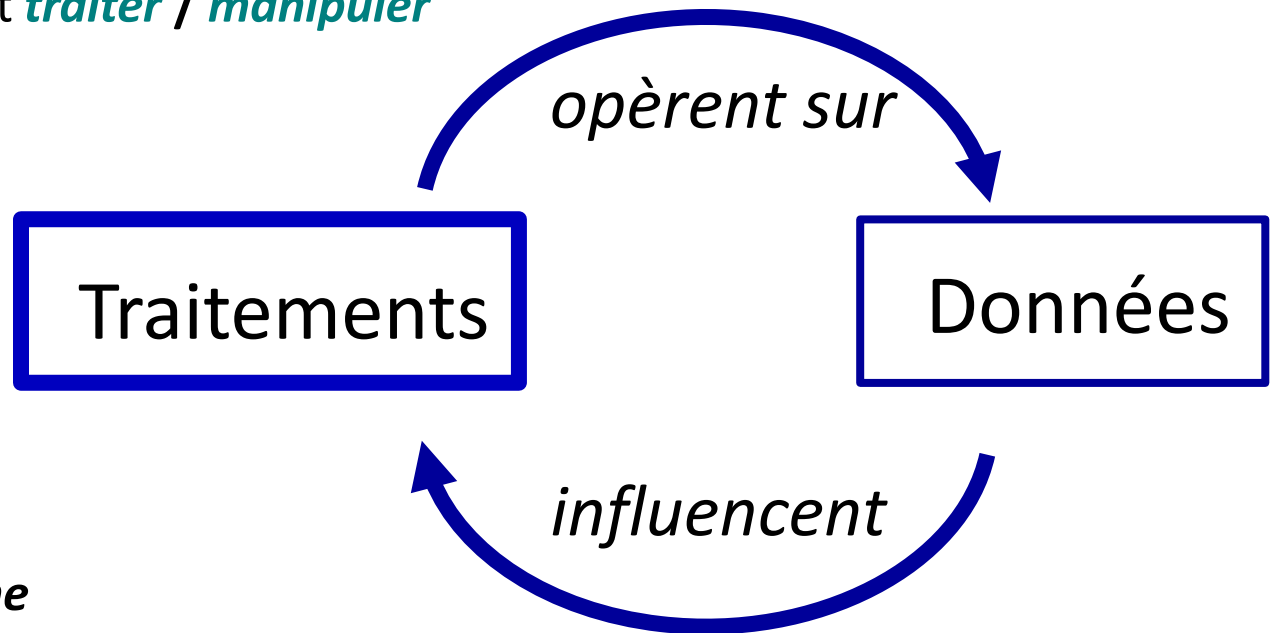
J.-C. Chappelier, J. Sam (slides) & R. Guerraoui (livre chap1)

## Objectifs de la leçon

Dans la leçon précédente, nous avons vu comment représenter les éléments d'un problème sous forme de **données** de différents *types* (nombres entiers, nombres à virgule flottante, caractères alphanumériques, etc...).

Une question est maintenant de savoir comment *traiter / manipuler* toutes ces **données**.

☞ C'est tout l'objet du **calcul** informatique.



Les objectifs de cette leçon sont de :

- Formaliser ces calculs : notion **d'algorithme**
- Présenter les «**composants**» des algorithmes
- Les illustrer sur la résolutions de quelques problèmes simples en partie inspirés par le cours précédent

## Plan

- Formaliser ces calculs : notion d'algorithme
- Présenter les « ingrédients de base » des algorithmes
- Illustrations sur quelques d'algorithmes

# Qu'est ce qu'un algorithme ?

## Algorithme ?

☞ moyen pour un humain de représenter la **résolution par calcul** d'un **problème** pour un autre humain ou pour une machine.

Les algorithmes existent depuis **bien avant les ordinateurs** :

Déjà dans l'Antiquité (e.g. *multiplication égyptienne, algorithme d'Euclide du PGCD*)

Origine du nom :

Mathématicien persan Al-Khwarizmi du 9<sup>e</sup> siècle, surnommé « le père de l'algèbre ».

# Formalisation de l'Informatique

## Formalisation des **traitements** : **algorithmes**

- ☞ distinguer formellement les bonnes séquences de traitements des mauvaises

## Formalisation des (ensembles de) **données** : **structures de données**

- ☞ Distinguer formellement les bonnes structuration des données des mauvaises. Cependant cet aspect est peu développé dans ce cours.
- ☞ Nous travaillerons essentiellement avec des variables et la notion de **liste**:

Une **liste** = un ***ensemble de données de taille connue  $N$*** .

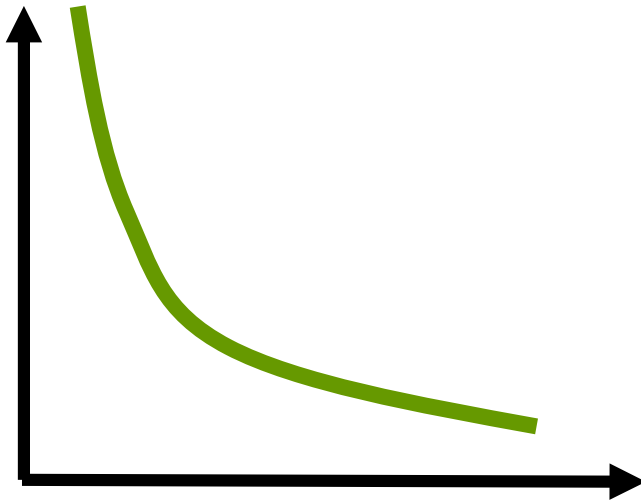
On accède à un élément de la liste **L** à l'aide d'un *numéro de **1 à  $N$*** , appelé **indice**.

Dans nos exemples on utilise les notations **L(i)** ou **L[i]** pour désigner le  **$i^{\text{ème}}$**  élément de la liste **L**.

La **conception** consiste à choisir les **bons algorithmes**  
*et les bonnes structures de données*  
pour résoudre un problème

Compromis fréquent entre *le coût calcul* de l'algorithme  
et *le coût mémoire* de la structure de donnée

**Coût calcul de l'algorithme** = nb d'opérations pour obtenir le résultat



**Coût mémoire de la structure de donnée**  
= espace mémoire nécessaire  
pour les données fournies en input,  
le résultat *ET les calculs intermédiaires*

Idéalement, si la mémoire était infinie et facile d'accès, de nombreux problèmes pourraient se mettre sous forme d'une table qu'il suffirait de consulter pour chaque combinaison des paramètres du problème fournie en entrée.

## Algorithme $\neq$ Programme

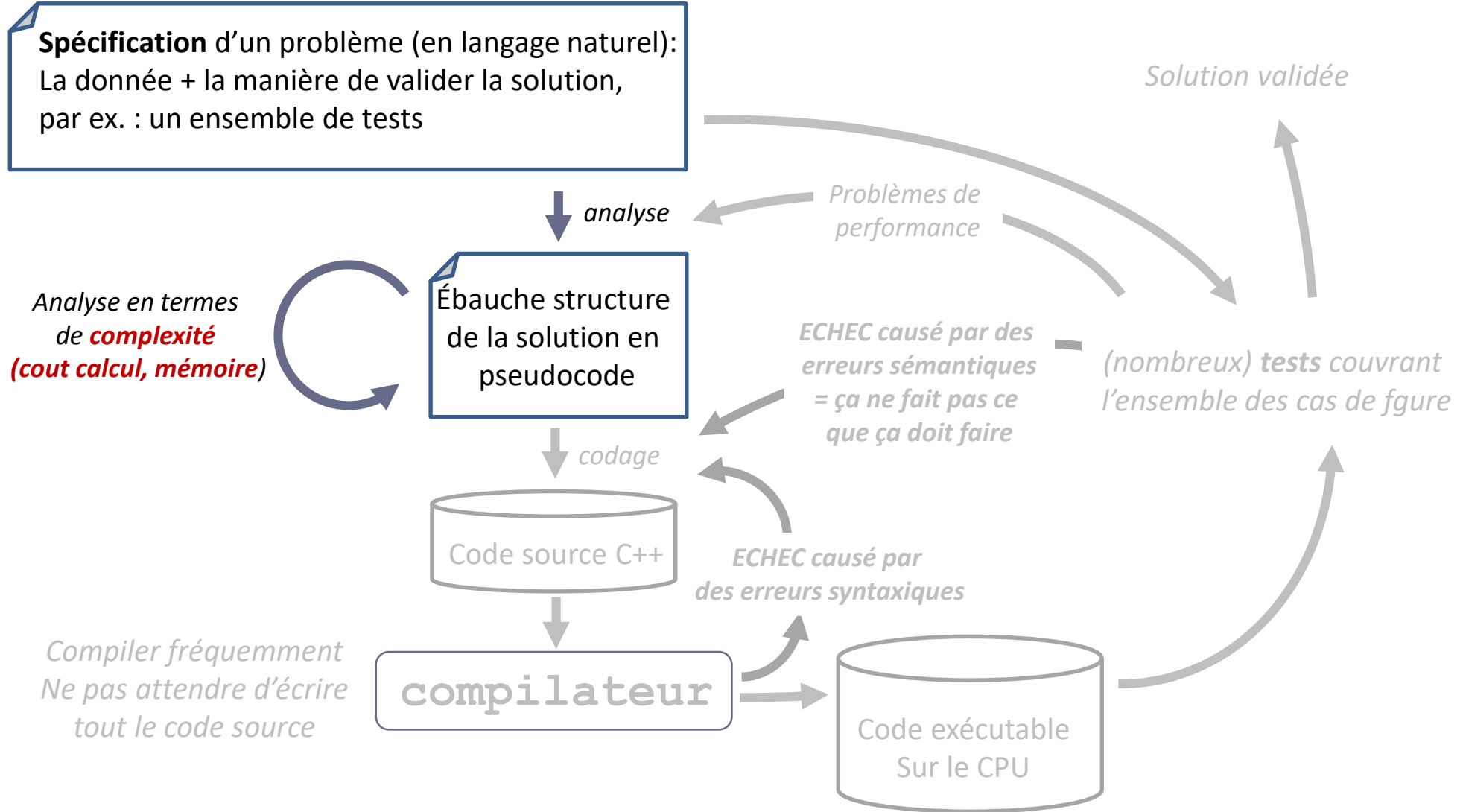
Un algorithme est **indépendant du langage de programmation** dans lequel on va l'exprimer **et de l'ordinateur** utilisé pour le faire tourner.

C'est une description *abstraite* des étapes conduisant à la solution d'un problème.

**algorithme** = partie conceptuelle d'un **programme** (indépendante du langage)

**programme** = implémentation (réalisation) de l'**algorithme**, dans un langage de programmation et sur un système particulier.

# Rappel: cycle de développement





# Algorithmes : introduction

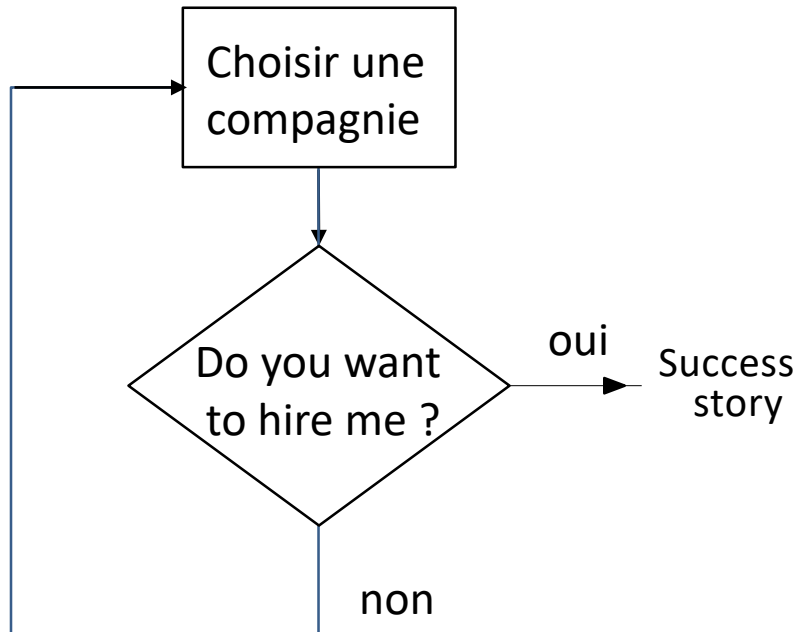


Do you want to  
hire me ?

☞ algorithme(s) / données ?..

# Algorithmes : introduction

« Do you want to hire me ? » : premier algorithme :



**Données :**

- Une personne cherchant un stage
- Ensemble de  $N$  compagnies

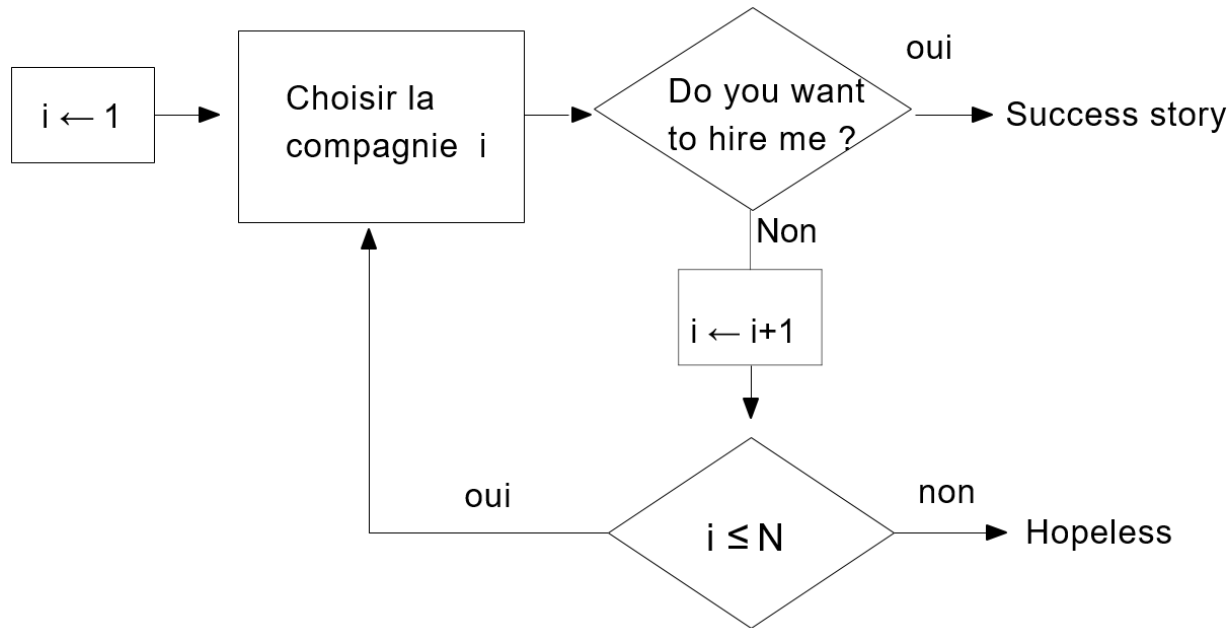
**Remarque :**

- Cette représentation graphique du flot de contrôle de l'algorithme s'appelle un ***organigramme***

👉 Il n'est pas garanti que l'algorithme puisse se terminer !

# Algorithmes : introduction

«Do you want to hire me ? » : deuxième algorithme :



L'algorithme se **termine** nécessairement  
(au pire  $N$  essais successifs)

**Données :**

- Une personne cherchant un stage
- Ensemble **ordonné** de  $N$  compagnies
  - Les données sont **structurées** ; chaque compagnie a un numéro entre 1 et  $N$
- Une variable  $i$  est utilisée pour compter les compagnies

## Qu'est ce qu'un algorithme ?

### Algorithme :

composition d'un **ensemble fini** d'opérations élémentaires bien définies (**déterministes**)

opérant sur un **nombre fini** de données

et effectuant un traitement bien défini :

- suite **finie** de règles à appliquer,
- dans un ordre déterminé,
- si possible, **se terminant** (i.e. arriver, en un nombre **fini** d'étapes, à un résultat, et cela quelles que soient les données traitées).

## Plan

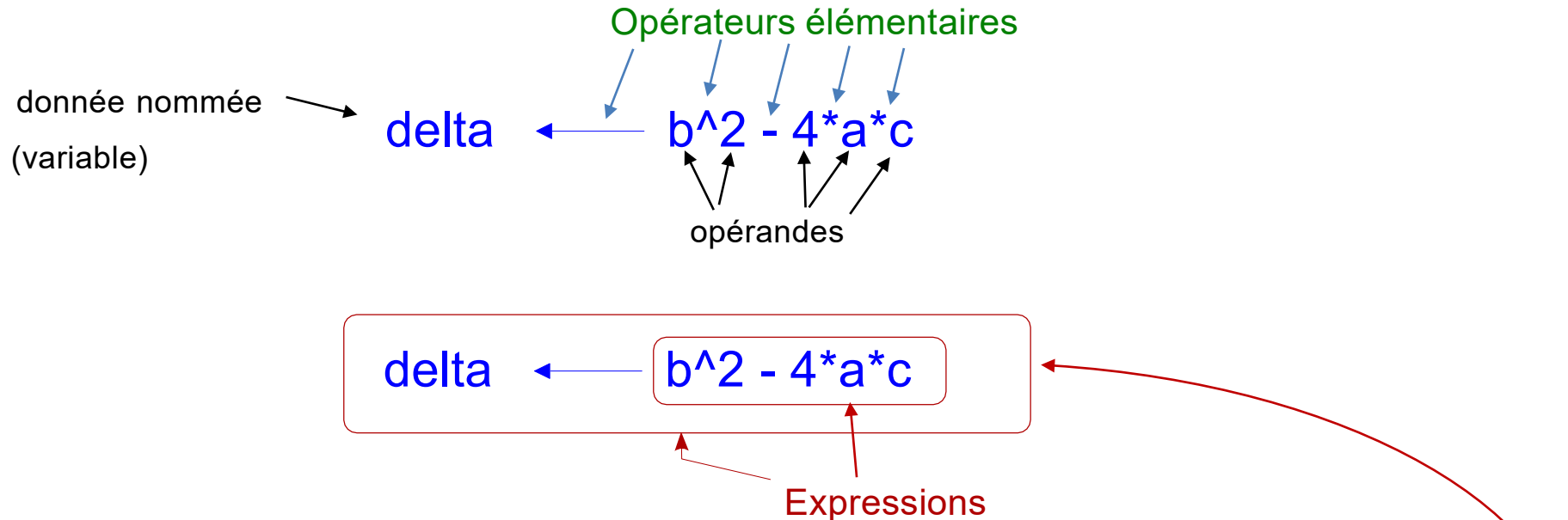
- Formaliser ces calculs : notion d'algorithme
- Présenter les « ingrédients de base » des algorithmes
- Illustrations sur quelques d'algorithmes

# Terminologie

Un algorithme travaille sur des données qu'il utilise et/ou modifie.

☞ il doit **mémoriser** ces données, en les associant à un *nom / identificateur* pertinent pour pouvoir les retrouver au moment où elles lui sont nécessaires.

Une donnée nommée est ce que l'on appelle une **variable** dans un algorithme.



Les **traitements** combinent une ou plusieurs **opérations** dans des **expressions**

L' **expression** correspondant à un pas de l'algorithme est appelée une **instruction**

## Instruction (non-) élémentaire

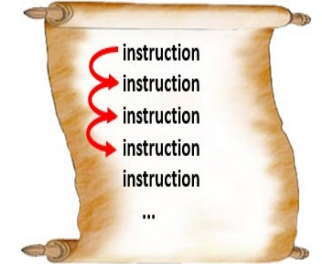
Une **instruction élémentaire** est une instruction dont le coût d'exécution est **constant** (ne dépend pas de la donnée que l'on manipule).

### Exemples :

- **instruction élémentaire d'affectation**:  
associer le résultat d'une expression à un nom (variable)  
$$\text{delta} \leftarrow b^2 - 4 * a * c$$
- **instruction élémentaire d'affichage** : utiliser le verbe **afficher**  
écrire la valeur d'une donnée, une variable ou expression dans le terminal
- **instruction non élémentaire** : compter le nombre de caractères contenus dans une phrase (dépend de la longueur de la phrase).

## Structures de contrôle

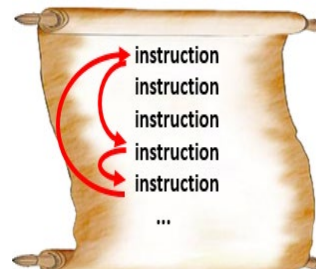
Une séquence linéaire d'instructions ne permet pas d'exprimer un traitement complexe des données (ex: conditionnel, répétitif).



### structures de contrôle

Une structure de contrôle sert à **modifier l'ordre séquentiel d'exécution** d'un programme (elle peut produire un *branchement* sur une instruction qui n'est pas celle qui suit dans l'algorithme)

 faire exécuter à la machine des tâches de façon *répétitive*, ou *en fonction de certaines conditions* (ou les deux).

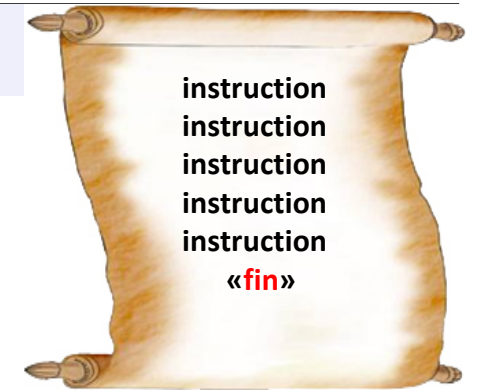




## Les 4 différentes structures de contrôle

### 1) la sortie = fin de l'algorithme: *Sortir : résultat (optionnel)*

La sortie *implicite* d'un algorithme est à la **fin** de la suite d'instructions.



Il est possible d'insérer une instruction de sortie de l'algorithme *explicitement* dans la suite d'instructions.

Si l'instruction de sortie n'est pas la dernière de l'algorithme, elle doit être associée à une condition sinon l'arrêt de l'algorithme est inconditionnel et ce qui suit n'est jamais exécuté.



La sortie *peut produire un résultat* mais ça n'est pas obligatoire.

Le *résultat* fourni en sortie peut être utilisé dans un autre algorithme

En général on associe l'instruction **sortir** à l'une des 3 autres structures de contrôle qui permettent de mettre en œuvre une action conditionnelle

## SpeakUp salle 93621: quel est le résultat de cet algorithme ?

Entrée: a, b, c entiers

But : affichage de la somme et du produit des entrées

somme  $\leftarrow a + b + c$

**sortir** somme

produit  $\leftarrow a * b * c$

**sortir** produit

- A. Algorithme correct ; affiche d'abord le produit puis la somme
- B. Algorithme incorrect; se termine en fournissant seulement la valeur du produit
- C. Algorithme correct ; affiche d'abord la somme puis le produit
- D. Algorithme incorrect ; se termine en fournissant seulement la valeur de la somme

## Algorithme corrigé

Entrée: a, b, c entiers

But: affichage de la somme et du produit des entrées

somme  $\leftarrow a + b + c$

**afficher** somme

produit  $\leftarrow a * b * c$

**afficher** produit

## Les 4 différentes structures de contrôle (2)

Ex: affichage de la parité d'un entier positif  $n$  .

L'opérateur **mod** calcule le *reste dans la division entière* de l'opérande gauche par l'opérande droit

### 2) les branchements conditionnels : *Si ... Alors ... Sinon ...*

*Si*  $(n \bmod 2) = 0$  *Alors*

afficher : ce nombre est pair

*Sinon*

afficher : ce nombre est impair

Le mot-clef *Si* est suivi d'une expression de type «**condition**» dont la valeur ne peut être que Vrai ou Faux (booléen)

On indique la ou les instructions contrôlées avec un décalage vers la droite = une *indentation*

### variante avec l'instruction *Sortir* ; le mot-clef *Sinon* devient inutile

*Si*  $(n \bmod 2) = 0$  *Alors*

afficher : ce nombre est pair

*Sortir*

Afficher : ce nombre est impair

→ Fin de l'algorithme si la condition est Vraie

→ Suite de l'algorithme si la condition est Fausse

## Les 4 différentes structures de contrôle (3)

### 3) les boucles conditionnelles :

*Tant que ...*

,

*Répéter ... Tant que...*

Le mot-clef *Tant que* est suivi d'une expression de type «**condition**» dont la valeur ne peut être que Vrai ou Faux (booléen)

```
lire un nombre entier n
k ← 0
Tant que n > 1
    n ← n / 2    //div entière
    k ← k + 1
sortir k
```

#### Répéter

```
demander une valeur positive
lire une valeur
Tant que la valeur n'est pas positive
... suite de l'algo ...
```

On exécute au moins une fois les instructions de la boucle

#### 4) les itérations : *Pour ... allant de ... à ...*

Lorsqu'on connaît le nombre de répétitions on privilégie l'usage de la boucle *Pour*

$$x = \sum_{i=1}^5 \frac{1}{i^2}$$



```
x ← 0
Pour i de 1 à 5
  x ← x + 1/i2
sortir x
```

**Implicitement** on considère que la variable *i* qui contrôle la boucle *Pour* est :

- Initialisée à la première valeur, ici **1**
- Testée systématiquement vis-à-vis de **5** pour décider si on entre dans la boucle ou pas
- **Incrémentée d'une unité** après l'exécution de l'instruction contrôlée

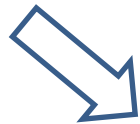
Exécution: Valeur initiale de x    |    valeur de i    |    valeur de x après exécution de l'instruction contrôlée

## On peut toujours faire des **itérations** en utilisant des **boucles conditionnelles**:

$$x = \sum_{i=1}^5 \frac{1}{i^2}$$



```
x ← 0
Pour i de 1 à 5
  x ← x + 1/i2
sortir x
```



```
x ← 0
i ← 1
Tant que i ≤ 5
  x ← x + 1/i2
  i ← i + 1
sortir x
```

## Algorithmes : conclusion

On attend d'un algorithme qu'il se termine, produise le résultat correct (propriété de sûreté) *pour toute entrée*.

Difficulté de l'Informatique (science) :  
assurer que l'algorithme est correct *pour toute entrée*.

On ne peut pas vérifier par des essais (empirisme) : on ne pourra jamais tester tous les cas

Mais il est possible d'organiser les tests de façon  
à couvrir de nombreux scénarios d'utilisations

Recherche en informatique:  vérification par preuves mathématiques

Importance d'un travail soigneux et mûrement réfléchi !



## Plan

- Formaliser ces calculs : notion d'algorithme
- Présenter les « ingrédients de base » des algorithmes
- Illustrations sur quelques algorithmes
  - Conversion représentation en virgule fixe vers décimal
  - Conversion base dix vers le binaire (nb entier)
  - Conversion base dix vers le binaire ( $0 \leq nb < 1$ )
  - Conversion représentation simple en virgule flottante vers décimal
  - Conversion base dix vers représentation simple en virgule flottante

## Ex : Conversion virgule fixe vers décimal

**Donnée** : Convertir un nombre binaire en virgule fixe représenté par une liste **P** de 32 valeurs binaires dont **nf** sont dédiés à la partie fractionnaire.

Le premier élément de la liste **P(1)** est le poids de la puissance  $2^{-nf}$ . Pour chaque élément suivant de la liste P, on augmente la puissance de la base de une unité.

### Conversion\_entier\_bin2dec

entrée : *nb de bits en partie fractionnaire nf*  
*liste P de 32 valeurs 0 ou 1*

sortie : *valeur représentée X*

$X \leftarrow 0$

**Pour** *i* de 1 à 32

$X \leftarrow X + P(i) * 2^{(i-1)-nf}$

**Sortir** *X*



## Ex2: Conversion d'un nombre entier décimal vers le binaire

**Donnée :** Convertir un entier positif décimal  $n$  en binaire jusqu'à un maximum de 32 bits. La conversion consiste à effectuer la division entière par la base 2 tant que le quotient est supérieur ou égal à la base. A chaque étape, **le reste de la division entière** fournit le *poids d'une puissance de la base* en commençant par le poids faible. On obtient ce *reste* avec L'opérateur **mod** .

**Question:** L'affichage immédiat des poids donne-t-il le bon résultat ?

**Réponse:** Non car le nombre binaire apparaît «à l'envers». On doit ranger chaque poids binaire dans un élément d'une liste **P** de 32 éléments binaires tel le poids de la puissance  $i-1$  de 2 est mémorisé dans l'élément  $P(i)$  de la liste P.

Dans une seconde étape on affiche les éléments mémorisés dans P en commençant par les poids fort.

Remarque: le pas est implicitement de **-1** pour  $k$

### Conversion\_entier\_dec2bin

entrée :  $n$ , entier naturel  
 $P$ , liste de 32 valeurs 0 ou 1  
sortie : aucune car l'algo affiche la valeur binaire de  $n$  ou un message d'erreur

$i \leftarrow 1$

**Répéter**

$P(i) = n \bmod 2$

$n \leftarrow n/2$

$i \leftarrow i + 1$

**Tant que**  $n \neq 0$  et  $i \leq 32$

**Si**  $n \neq 0$  et  $i > 32$

Afficher :  $n$  est trop grand

**Sinon**

**Pour**  $k$  de  $(i - 1)$  à 1

afficher :  $P(k)$



## Ex3: Conversion décimal vers binaire : $0 \leq a < 1$

**Donnée** : Convertir un nombre fractionnaire positif décimal **a** en binaire jusqu'à un maximum de 32 bits.

La conversion consiste à une multiplication de la partie fractionnaire par la base 2 tant qu'elle n'est pas nulle. A chaque étape, la partie entière est le poids d'une puissance de la base. L'algorithme produit le poids fort en premier. De ce fait on peut procéder à l'affichage immédiatement, sans passer par une mémorisation des bits.

**Illustration**: supposons que **a** ait cette représentation binaire:  $0.b_w b_x b_y b_z$  où chaque lettre *b* désigne la valeur d'un bit, 0 ou 1.

Si ce motif binaire est multiplié par 2, il se décale d'un cran vers la gauche: supposons  $X = b_w . b_x b_y b_z$

La partie entière notée **PartieEntière(X)** donne  $b_w$   
L'algorithme continue avec  $0. b_x b_y b_z$

### Conversion\_fract\_dec2bin

entrée : *a*, dans l'intervalle  $[0, 1[$   
sortie : aucune car l'algo affiche la valeur binaire de *a* et éventuellement un message d'avertissement

afficher: 0.

$i \leftarrow 1$

**Répéter**

$a \leftarrow 2 * a$

$b \leftarrow \text{PartieEntière}(a)$

afficher : b

$a \leftarrow a - b$

$i \leftarrow i + 1$

**Tant que**  $a > 0$  et  $i \leq 32$

**Si**  $i > 32$

Afficher : ce résultat est  
une approximation



## Ex : Conversion virgule flottante vers décimal

**Donnée** : Convertir la valeur binaire d'une liste F contenant 4 valeurs binaires pour la représentation en virgule flottante du cours M1.L1 slides 36-37.

Comme indiqué sur le dessin à droite:

- F(4) et F(3) sont respectivement les poids des puissances 1 et 0 de l'exposant.
- F(2) et F(1) sont respectivement les poids des puissances -1 et -2 de la mantisse.

Si l'**exposant** a la valeur minimum de **0**, on utilise la **forme dénormalisée**:

$$X = 2^1 * 0, \text{mantisse}$$

sinon la **forme normalisée**.

$$X = 2^{\text{exposant}} * 1, \text{mantisse}$$

### Conversion\_virgule\_flottante\_vers\_dec

entrée : liste F de 4 valeur binaires

sortie : la quantité représentée X

$$\text{exposant} \leftarrow 2 * F(4) + F(3)$$

$$\text{mantisse} \leftarrow 0.5 * F(2) + 0.25 * F(1)$$

**Si** exposant = 0

// forme dénormalisée

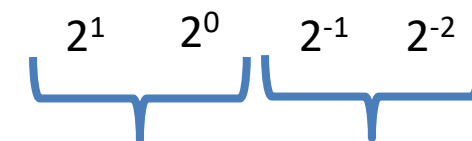
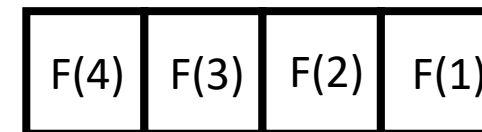
Sortir : 2 \* mantisse

**Sinon**

// forme normalisée

Sortir :  $2^{\text{exposant}} * (1 + \text{mantisse})$

Liste F :



exposant

mantisse

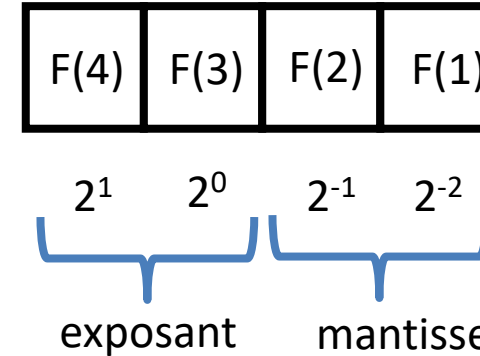




## Ex : Conversion décimal vers virgule flottante

**Donnée :** Convertir la valeur décimale positive  $X$  comprise dans l'intervalle  $[0, 16[$  vers la représentation de l'exemple simple du cours M1.L1 slides 36-37. On utilise la liste  $F$  pour stocker le résultat (dessin).

Liste  $F$  :



**Méthode:** on doit distinguer le cas particulier de 0 et les nombres correspondants à la forme dénormalisée (slide 34) pour la valeur minimum de l'exposant. Ces nombres sont dans l'intervalle  $[0, 2^1[$  ; on pose  $p=1$  dans la suite de la méthode.

Pour les autres nombre dans l'intervalle  $[2, 16[$ , on calcule  $p = \text{PartieEntière}(\log_2(X))$  pour trouver dans quelle plage de puissance de 2 se trouve  $X$ .

On en déduit l'**exposant** : 0 pour la forme dénormalisée  
 sinon conversion de  $p$  en binaire

On obtient la **mantisse** en convertissant en binaire la partie fractionnaire de  $v = X/2^p$ .  
 Pour la forme dénormalisée,  $v$  est dans l'intervalle  $] 0, 1[$   
 Pour la forme normalisée,  $v$  est dans l'intervalle  $[ 1, 2[$ .

## Ex : Conversion décimal vers virgule flottante

### Conversion\_dec\_vers\_vflottante

entrée : la quantité décimale X dans l'intervalle [0, 16[

sortie : la liste F du motif binaire du cours M1.L1

```
Si  $X = 0$  // zéro est représenté
    Sortir :  $F \leftarrow \{0, 0, 0, 0\}$  // par le motif binaire nul
Sinon Si  $X < 2^1$  // forme dénormalisée
     $p \leftarrow 1, F(4) \leftarrow 0, F(3) \leftarrow 0$ 
Sinon // forme normalisée
     $p \leftarrow \text{PartieEntière}(\log_2(X))$ 
     $F(4) \leftarrow p/2$  // division entière
     $F(3) \leftarrow p \bmod 2$ 

 $v \leftarrow X / 2^p$  // division sur les réels
 $m \leftarrow v - \text{PartieEntière}(v)$  // obtient la partie fractionnaire
 $b \leftarrow \text{PartieEntière}(2 * m)$ 
 $F(2) \leftarrow b$ 
 $m \leftarrow 2 * m - b$  // obtient la partie fractionnaire
 $F(1) \leftarrow \text{PartieEntière}(2 * m)$ 
Sortir: F
```



## Ce que j'ai appris aujourd'hui

***Dans cette leçon, vous avez***

... appris ce qu'est un **algorithme** et ses principaux constituants

Expressions

Structures de contrôle

*Si alors Sinon,*

*Tant que,*

*Répéter .... Tant que*

*Pour i de ..à ..*

*Sortir*

***👉 Vous pouvez maintenant :***

... construire des algorithmes simples pour des problèmes simples

La prochaine leçon présentera :

- la notion de **complexité** d'un algorithme
- Deux grandes famille d'algorithmes: la **recherche** et le **tri**
- La *stratégie descendante (top-down)* de conception