R107
Fondamentaux de la programmation

Ismail Bennis
Ismail.bennis@uha.fr
MCF, IUT de Colmar, Département Réseaux & Télécoms, bureau N°006
34 rue du Grillenbreit - 68000 Colmar Cedex

R107

- **Compétence ciblée** : **RT3-Programmer Niveau 1**
  - **AC0311** : Utiliser un système informatique et ses outils
  - AC0312 : Lire, exécuter, corriger et modifier un programme
  - AC0313 : Traduire un algorithme, dans un langage et pour un environnement donné
  - AC0316 : S'intégrer dans un environnement propice au développement et au travail collaboratif
- **Mots-clés** : Algorithmes, Langages de programmation, Python, Suivi de versions.
- **Coef**. 22
- **Volume horaire** : 39h, dont 24h de TP
- **Evaluation** : DS + 2 Exam. partique

Plan du module

01

- **Algorithmique**
  - Variables, types de base (nombres, chaînes, listes/tableaux).
  - Structures de contrôle : tests, répétitions.
  - Fonctions et procédures.
  - Portée des variables.

02

- **Langage de programmation**

03

- **Introduction à Python**
  - Prise en main d'un IDE.
  - Prise en main de bibliothèques, modules, d'objets existants.
  - Manipulation de fichiers texte.
  - Interaction avec le SE et la CLI : arguments, lancement de commandes.
  - Suivi de versions (git, svn).

www.uha.fr

UNIVERSITÉ HAUTE-ALSACE

L’algorithmique

L’algorithmique

- **Algorithme** n. m. 1554 latin médiév. Algoritmus latinisé de l’arabe Al khawarizmi (cf. algèbre) : suite finie et séquentielle de règles que l’on applique à un nombre fini de données permettant de résoudre des classes de problèmes semblables.
- Ensemble des actions nécessaires à l’accomplissement d’une tâche.
- Description d'une action complexe, au moyen d'actions élémentaires et de règles de composition de ces actions.
- Similaire à une recette de cuisine, un mode d’emploi, etc.
- Objectif :
    - **Faire faire** quelque chose à l’ordinateur.
    - **Réutiliser** un traitement complexe.
    - Résoudre un problème pour l’appliquer à **diverses données**.

L’algorithmique

- **Programme** : c'est le codage d'un **algorithme** dans un **langage de programmation** compréhensible par un ordinateur.
- Le langage utilisé n'a pas d'importance : un bon algorithme peut être traduit dans n'importe quel langage de programmation.
- **Algorithme vs Programme :**
  - ✓ L’algorithme décrit la suite d’actions ou de calculs à faire pour effectuer la tâche.
    - Il s’écrit en **pseudo-code**.
    - Il est suffisamment générique pour être traduit en un langage de programmation.
  - ✓ Le programme traduit cette suite d’actions en langage d’ordinateur.
    - Le programme va être interprété par un ordinateur.
    - À ce titre, il doit **respecter des contraintes** de formes plus précises que l’algorithme.
    - Il peut être illisible par un humain

Les étapes de construction d'un programme

1) Question (problème) ➔ 2) Enoncé (cahier des charges) ➔ 3) Résolution (algorithme) ➔ 4) Mise en œuvre (programme)

1) Construire le cahier des charges : de la question à l'énoncé.

- Lorsqu'un problème est posé, la première chose à faire est de l'énoncer précisément, en décrivant :
  - ✓ les données du problème (l'information fournie en entrée)
  - ✓ les résultats que l'on souhaite obtenir (l'information désirée en sortie)
  - ✓ la relation entre les données et les résultats.
- Il faut aussi décrire comment les données doivent être obtenus, et sous quelle forme le résultat doit être présenté.
- Chaque donnée du problème (entrée ou résultat) devra être :
  - ✓ désigné par un nom significatif pour le distinguer des autres **objets**.
  - ✓ typé de manière précise.

Les étapes de construction d'un programme

Exemple :

Question : Ecrire un programme permettant de calculer le prix total d'une commande de plusieurs éléments d'un même produit en tenant compte des frais de transports.

Cahier des charges :

Données :

PRIX_UNITAIRE : de type réel, représentant le prix unitaire TTC d'un produit, fourni par l'opérateur.

QUANTITE : de type entier, représentant la quantité de produits acheté.

TRANSPORT : de type réel, représentant le coût du transport, fourni par l'opérateur

Résultat :

PRIX : de type réel, représentant le prix total, retourné à l'opérateur

Relation :

PRIX = PRIX_UNITAIRE * QUANTITE + TRANSPORT

Les étapes de construction d'un programme

2) Construire l'algorithme détaillé : de l'énoncé à la résolution.

- Il s'agit de décrire de manière détaillée **l'enchaînement des opérations** élémentaires qui mènera au résultat spécifié dans le cahier des charges.
- Lorsque le problème est complexe, il est nécessaire de décomposer les actions en plusieurs niveaux, du plus général au plus particulier.
- Lorsqu'il s'agit d'un algorithme à plusieurs niveaux, chaque sous-algorithme peut être ainsi vérifié individuellement
- Il est nécessaire de faire **dérouler un algorithme** à la main (sur papier, dans sa tête), pour vérifier son bon fonctionnement. Cette vérification est l'occasion de bâtir des **jeux d'essais** qui pourront être réutilisés pour valider le programme final.

Les étapes de construction d'un programme

Algorithme principal :

Début

Demander PRIX_UNITAIRE avec "Quel est le prix unitaire ?"

Demander QUANTITE avec "Quel est la quantité ?"

Demander TRANSPORT avec "Quel est le coût de transport ?"

Mettre dans PRIX le résultat de PRIX_UNITAIRE * QUANTITE + TRANSPORT

Ecrire "Le Prix total est :" PRIX

Fin

Algorithme de "Demander VARIABLE avec QUESTION "

Début

Ecrire QUESTION

Lire VARIABLE

Fin

- Les actions élémentaires utilisées sont **Lire**, **Ecrire**, l'affectation (**Mettre**) et les **opérations arithmétiques ou logiques**.
- Toutes les actions sont exécutées **séquentiellement** (l'une après l'autre), sans retour en arrière, entre les mots Début et Fin

Les étapes de construction d'un programme

3) Construire le programme : de la résolution à la mise en œuvre.

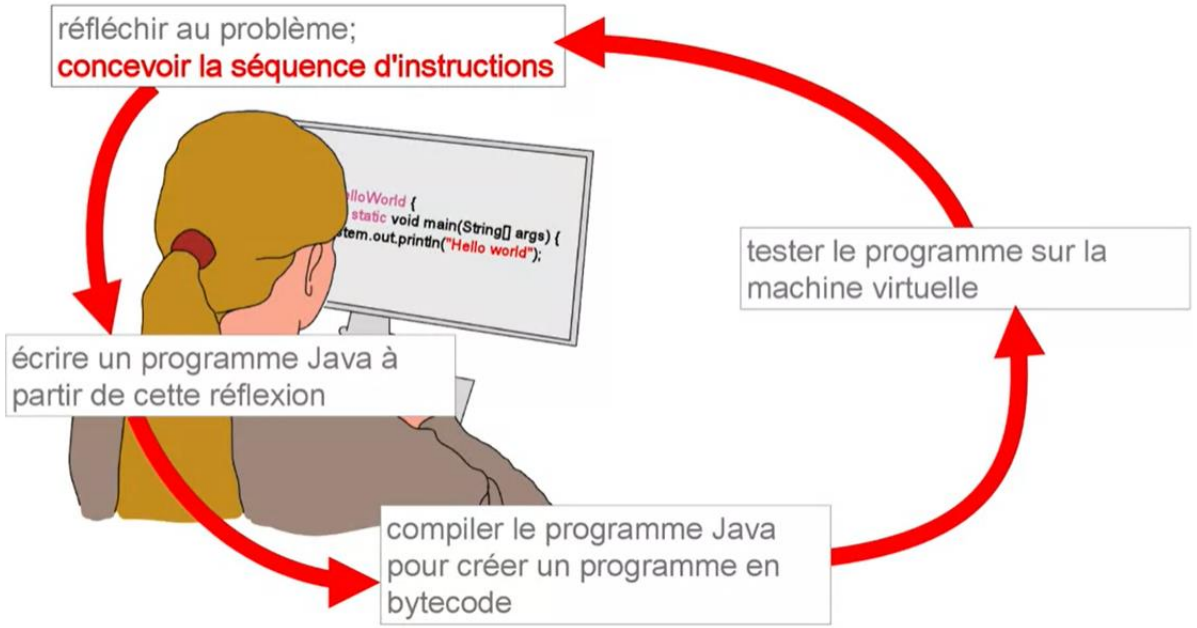
Lorsque l'algorithme est écrit, la construction d'un programme est une simple traduction dans le langage de notre choix.

```
float DEMANDER(char* QUESTION)  /* Demande une valeur à l'opérateur */
  {
        float VARIABLE;
        printf("%s",QUESTION);
        scanf("%f",&VARIABLE);
        return VARIABLE ;
  }

main()
  {
        float     PRIX,PRIX_UNITAIRE,TRANSPORT;
        int       QUANTITE;

        /* Saisie des donnees*/
        PRIX_UNITAIRE=DEMANDER("Quel est le prix unitaire ?");
        QUANTITE=(int)DEMANDER("Quel est la quantite ?");
        TRANSPORT=DEMANDER("Quel est le coût de transport ?");

        /* Calcul du coût */
        PRIX=PRIX_UNITAIRE*QUANTITE+TRANSPORT;

        /* edition des resultats */
        printf("Le prix total est : %f",PRIX);
  }
```

Programme en C

L'algorithmique : remarques

- Pour réussir son algorithme :
  - Il faut avoir une écriture **rigoureuse**
  - Il faut avoir une écriture soignée : par exemple respecter **l'indentation**
  - Il est nécessaire de **commenter** les algorithmes
  - Il existe **plusieurs solutions algorithmiques** à un problème posé
  - Il faut rechercher l'efficacité de ce que l'on écrit



L'algorithmique : notions de base

1. Données

En algorithmique, toute donnée est définie par :

- son **nom**: désigne la donnée dans l'algorithme,
- son **type**: désigne le domaine de valeurs de la donnée,
- sa **nature**: variable ou constante.

Type	Domaine
booléen	$\{faux, vrai\}$
caractère	Symboles typographiques
entier	$\mathbb{Z}$
réel	$\mathbb{R}$

2. Operateurs

Un operateur est une fonction définie par

- le nombre de variables d'entrée,
- sa position: l'operateur peut être préfixe (devant), infixe (milieu) ou postfixe (derrière),
- le type de ses entrées et celui de sa sortie.

- Arithmétiques (entiers): toutes les entrées sont des **entiers** et la sortie est un **entier**.

Nom	Symbole
addition	$+$
soustraction	$-$
multiplication	$\times$
division entière	$/$
reste	mod (%)
inversion de signe	$-$

- Arithmétiques (réels): au moins une entrée est un **réel** et la sortie est un **réel**.

Nom	Symbole
addition	$+$
soustraction	$-$
multiplication	$\times$
division	$/$
inversion de signe	$-$

- Comparaisons: les deux entrées sont des **entiers**, **caractères** ou **réels**. La sortie est un **booléen**.

Nom	Symbole
est égal à	$=$
est plus petit que	$<$
est plus grand que	$>$
est plus petit ou égal à	$\leq$
est plus grand ou égal à	$\geq$

- Logiques: toutes les entrées sont des **booléens** et la sortie est un **booléen**.

Nom	Symbole
conjonction	**et**
disjonction	**ou**
négation	**non**

L'algorithmique : notions de base

3. Expressions

- Une expression est une composition d'opérations dont l'ordre est spécifié par les parenthèses.
- Le type d'une expression est donné par le type de sa valeur de sortie
- Exemple: supposons que x, y, z soient des entiers.
  - (x > 0) et (y < 0) est une expression booléenne
  - (x + y)/z est une expression entière

4. Instructions

Une instruction est une action à accomplir par l'algorithme. Les quatre **instructions de base** sont : la **déclaration** (mémoire), l'assignation ou **affectation** (calcul), la **lecture** (entrées) et **l'écriture** (sorties).

Instruction	Spécification
Déclaration	*type variable*
Assignation	*variable* ⟵ *expression*
Lecture	**lire** *variable*
Ecriture	**écrire** *expression*

- saisie : instruction lire()
  exemple : a ← lire()
  Rempli la variable 'a' à l'aide d'une saisie clavier
- affichage : instruction affiche()
  exemple : affiche(" bonjour ", a)
  Affiche le contenu de la variable a précédée de « bonjour »

L'algorithmique : notions de base

4. Instructions

- Exemple de déclaration et d'affectation :

```
Début
    // déclaration de constantes
    réel pi ← 3.14158
    entier c ← 299792458
    // déclaration de variables
    réel x,y
    entier m,n

    x ← 2.0          // x prend la valeur 2.0
    y ← 4.0          // y prend la valeur 4.0
    y ← y + 1        // y est incrémenté de 1 (nouvelle valeur 5.0)
    x ← y            // prend la valeur de y c'est-à-dire 5.0
fin
```

L'algorithmique : notions de base

4. Instructions

- Exemple des instructions d'**entrées-sorties** :

```
Début
    // déclaration de variables
    réel distance, temps, vitesse

    afficher "Distance :"
    lire distance ;


    afficher "Temps : "
    lire temps ;

    vitesse ← distance / temps
fin
```

// L'algorithme indique à l'écran d'afficher la chaîne 'Distance :'

// L'algorithme lit la valeur entrée au clavier et l'affecte à la variable 'distance'

// L'algorithme indique à l'écran d'afficher la chaîne 'Vitesse' : suivie de la valeur de la variable 'vitesse'

- L'instruction **lire x** permet de recevoir une valeur entrée au clavier et de l'affecter à x
- L'instruction **afficher x** permet d'afficher à l'écran la valeur de x

L'algorithmique : notions de base

5. Blocs

Un bloc est une séquence d'instructions identifiée par une barre verticale ou bien ils ont le même niveau d'**indentation**

Exemple: permutation de valeurs

```
début
    entier a, b, temp
    lire a, b
    temp ⟵ a
    a ⟵ b
    b ⟵ temp
    afficher a, b
fin
```

6. L'instruction de test "si alors" (ou "if then")

Dans l'instruction **si condition alors bloc**, la condition est une expression booléenne, et le bloc n'est exécuté que si la condition est vraie.

Exemple: valeur absolue

```
début
    réel x, y
    lire x
    y ← x
    si y < 0 alors
        y ← −y
    afficher y
fin
```

L'algorithmique : notions de base

7. L'instruction de test "si alors sinon" (ou " if then else")

Dans si **condition alors bloc 1 sinon bloc 2**, la condition est une expression booléenne. Le bloc 1 est exécutée si la condition est vraie ; le bloc 2 est exécuté si la condition est fausse

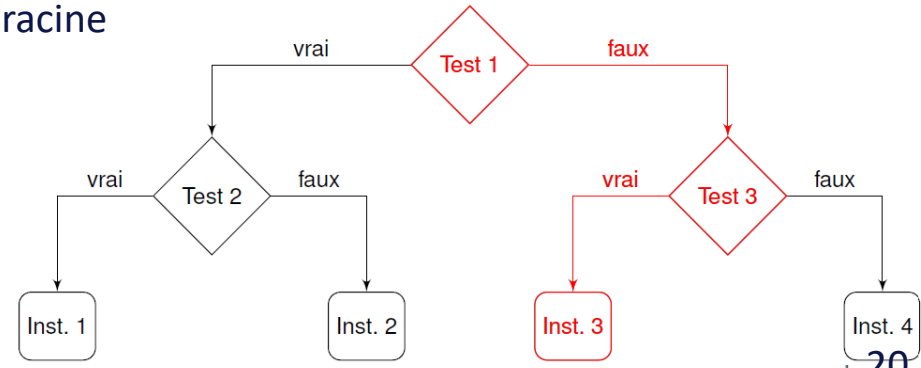
Exemple: racine carrée

```
début
    réel x, y
    lire x
    si x ≥ 0 alors
        y ← sqrt(x)
        afficher y
    sinon
        afficher "Valeur indéfinie"
fin
```

8. Arbres de Décision

Une représentation graphique des **tests imbriqués**. Chaque sommet interne représente la condition d'un "si alors (sinon)« et chaque feuille représente un bloc d'instructions.

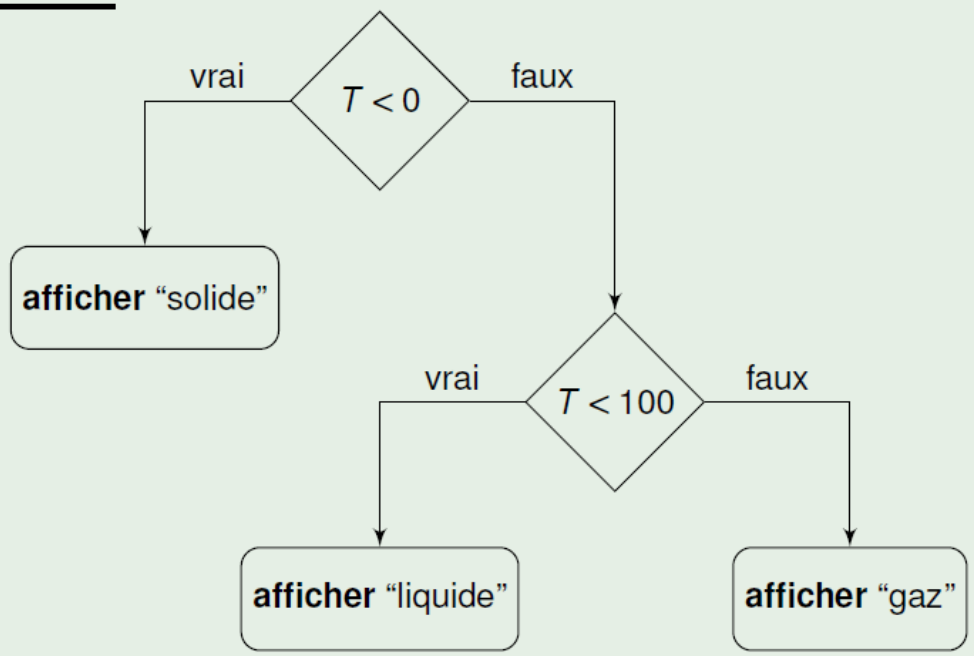
L'exécution de l'algorithme est un chemin depuis la racine jusqu'à l'une des feuilles.



L'algorithmique : notions de base

8. Arbres de Décision

```
Algorithme : étatDeLeau
variable
    réel T

début
    lire T
    si T < 0 alors
        afficher "solide"
    sinon
        si T < 100 alors
            afficher "liquide"
        sinon
            afficher "gaz"
        fin
    fin
fin
```



L'algorithmique : notions de base

9. L'instruction de test "suivant cas" (ou " switch case")

Dans l'instruction **suivant condition cas où v1 bloc 1 cas où v2 bloc 2** . . . , la condition est une expression pouvant prendre plusieurs valeurs v1, v2, ... Selon la valeur de la condition, le bloc du cas correspondant est exécuté.

Exemple: choix de menu

```
début
    entier menu
    lire menu
    suivant menu faire
        cas où 1
            afficher "Menu enfants"
        cas où 2
            afficher "Menu végétarien"
        autres cas
            afficher "Menu standard"
fin
```

L'algorithmique : notions de base

10. L'instruction de boucle "pour" (ou "for ")

- L'instruction pour est utilisée lorsque le nombre d'**itérations** est connu à l'avance : elle initialise un compteur, l'incrémente après chaque exécution du bloc d'instructions, et vérifie que le compteur ne dépasse pas la borne supérieure.

- Exemple: Somme des entiers de 1 à n

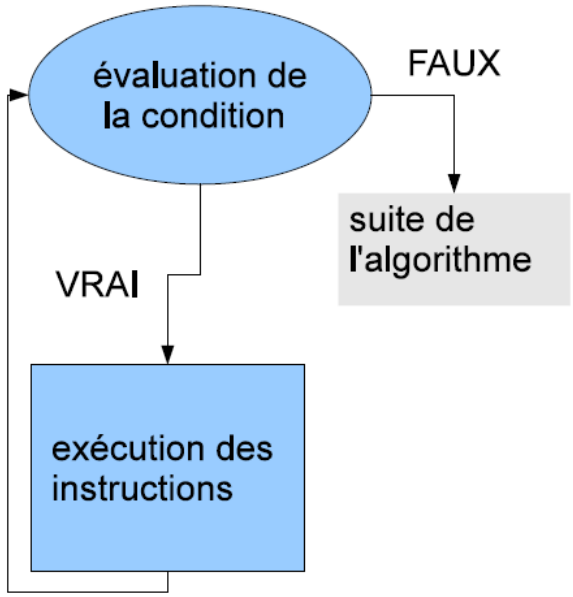
```
début
    entier n, s, i          // Supposons que n = 10
    lire n
    s ← 0                   // La somme s est initialisée à 0
    pour i de 1 à n faire   // La boucle est exécutée 10 fois
        s ← s + i           // La somme s prend la valeur 0+1+...+10
    afficher s
fin                         // La somme finale est 55
```

- Attention : dans une instruction **pour**, le compteur ne doit jamais être modifié par le bloc d'instructions

L'algorithmique : notions de base

11. L'instruction de boucle "tant que" (ou "while")

La boucle tant que est utilisée lorsque le nombre d'itérations n'est pas connu à l'avance. Elle exécute le bloc d'instructions tant que la condition reste vraie.



Exemple: Somme des entrées saisies par l'utilisateur (version "tant que")

```
début
    entier n ← 1, s ← 0
    tant que n ≠ 0 faire
        afficher "Entrer un entier (0 pour arrêter) : "
        lire n
        s ← s + n
    afficher s
fin
```

- N.B. Il est possible d'imbriquer les deux types de boucle (while et for)

L'algorithmique : notions de base

11. L'instruction de boucle "tant que" (ou "while")

- **Première itération :**

```
début
    entier n ← 1, s ← 0                    // Comme n = 2, la condition est vraie
    tant que n ≠ 0 faire
        afficher "Entrer un entier (0 pour arrêter) : "
        lire n                             // Supposons que la saisie de n soit 2
        s ← s + n                          // La somme s prend la valeur 0+2 = 2
    afficher s
fin
```

- **Deuxième itération :**

```
début
    entier n ← 1, s ← 0                    // Comme n = 0, la condition est fausse
    tant que n ≠ 0 faire
        afficher "Entrer un entier (0 pour arrêter) : "
        lire n                             // Supposons que la saisie de n soit 0
        s ← s + n                          // La somme s prend la valeur 2+0 = 2
    afficher s
fin                                        // La somme finale est 2
```

L'algorithmique : notions de base

12. L'instruction de boucle "répéter jusqu' à" (ou "do while ")

- La boucle répéter jusqu' à est utilisée lorsque le nombre d'**itérations** n'est pas connu à l'avance, et qu'il faut lancer au moins une exécution du bloc d'instructions. Elle exécute le bloc jusqu' à ce que la condition d'arrêt devienne vraie.

- Exemple: Somme des entrées saisies par l'utilisateur (version "répéter jusqu' à")

```
début
    entier n, s ← 0
    répéter
        lire n
        s ← s + n
    jusqu'à n = 0
    afficher s
fin
```

- **Comparaisons**

✓ La boucle « **pour i de x à y faire** instruction » exécute l'instruction exactement y-x +1 fois.

✓ La boucle « **tant que** condition **faire** instruction » exécute l'instruction jusqu'à ce que la condition soit fausse. L'instruction peut donc être exécutée **zero** fois si la condition est initialement fausse.

✓ La boucle « **répéter** instruction **jusqu'à** condition » exécute l'instruction jusqu'à ce que la condition soit fausse. L'instruction est donc exécutée au moins une fois.

L'algorithmique : notions de base

- **Instructions break**
- L'instruction break permet d'arrêter la boucle (for ou while) avec un test (if).
- Souvent utilisée avec un **while (true)** (boucle infinie)
- Exemple: somme des entrées saisies par l'utilisateur (version avec break)

```
début
    entier n ← 1, s ← 0
    tant que n ≠ 0 faire
        afficher "Entrer un entier (0 pour arrêter) : "
        lire n
        s ← s + n
    afficher s
fin
```

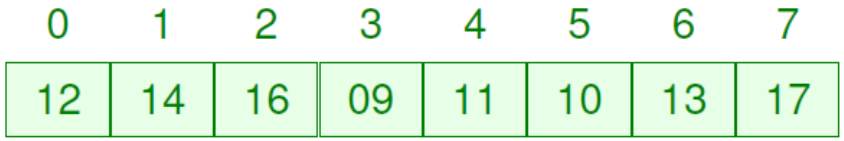
```
début
  entier n, s ← 0
  Tant que Vrai faire
    lire n
    si (n == 0)
          break
    s ← s +n
  afficher s
fin
```

- Une **boucle infinie** est, en programmation informatique, une boucle dont la condition de sortie n'a pas été définie ou ne peut pas être satisfaite.

L'algorithmique : notions de base

12. Tableaux Unidimensionnels

- Un tableau (statique unidimensionnel) est une séquence de données du même type accessibles par leur index. Il est défini par:
  - son **nom**,
  - le **type** de ses éléments, et
  - sa **taille** ou le nombre de ses éléments.



Un tableau de 8 entiers

- Le premier **index** d'un tableau de N éléments est **0** et le dernier index est **N - 1**.
- Les seules opérations possibles sont la déclaration, l'initialisation (déclaration avec valeurs initiales) et l'accès à ses éléments.

Opération	Spécification	Exemple
Déclaration	*type nom* [*taille*]	**entier** *tab*[10]
Initialisation	*type nom* [*n*] $\leftarrow \{v_1, \cdots, v_n\}$	**caractère** *voyelles*[5] $\leftarrow$ {'a','e','i','o','u','y'}
Accès	*nom* [*index*]	*voyelles*[*i*]

L'algorithmique : notions de base

13. Tableaux multidimensionnels

- Un tableau statique de dimension **d** est une séquence de tableaux de dimension **d - 1**. En particulier, une matrice est un tableau de dimension 2. Comme pour les tableaux statiques, les opérations possibles sont la déclaration, l'initialisation et l'accès à ses éléments.

Opération	Spécification	Exemple
Décl.	*type nom* [*rangées*][*colonnes*]	**réel** *matrice*[4][4]
Init.	*type nom* [*m*][*n*] $\leftarrow \{\{v_{11}, \cdots, v_{1n}\}, \cdots, \{v_{m1}, \cdots, v_{mn}\}\}$	**réel** *unité*[2][2] $\leftarrow \{\{1,0\},\{0,1\}\}$
Accès	*nom* [*rangée*][*colonne*]	*matrice*[*i*][*j*]

14. Chaîne de caractères

Une chaîne de caractères est un tableau dynamique unidimensionnel composé de caractères ascii. Il est possible de faire des opérations de comparaison lexicographique.

Opération	Spécification	Exemple
Déclaration	**chaîne** *nom*	**chaîne** *c*
Initialisation	**chaîne** *nom* $\leftarrow$ *constante chaîne*	**chaîne** *c* $\leftarrow$ "Bonjour"
Copie	$nom_1 \longleftarrow nom_2$	$c \leftarrow d$
Accès aux éléments	*nom*[*index*]	*c*[*i*]
Accès à la taille	**longueur**(*nom*)	**longueur**(*c*)
Test de la chaîne vide	**vide**(*nom*)	**si**(**vide**(*c*)) **alors** ...
Concaténation	$+$	$c \leftarrow c + d$
Comparaisons	$\leq, <, =, \neq, >, \geq$	**si**($c \neq d$) **alors** ...

L'algorithmique : notions de base

15. Algorithmes de recherche

- Algorithmes permettant la recherche d'un objet dans un ensemble en utilisant des boucles "**tant que**": la recherche s'arrête dès que l'élément est trouvé.
- Exemple: recherche un entier dans un vecteur non trié; si la valeur recherchée est présente alors l'algorithme retourne son index, sinon il retourne la taille du vecteur

```
entier rechercher(vecteur d'entiers tab, entier x)
début
    booléen trouvé ← faux
    entier i ← 0
    tant que (i < longueur(tab)) et (non trouvé) faire
        trouvé ← tab[i] = x
        si non trouvé alors i ← i + 1
    retourner i
fin
```

16. Algorithmes de Tri

- Les algorithmes de tri simple (insertion, sélection) utilisent deux boucles "pour"

```
triParSelection(vecteur d'entiers tab)
début
    entier i, j
    pour i de 0 à longueur(tab) − 2 faire
        pour j de i + 1 à longueur(tab) − 1 faire
            si tab[j] < tab[i] alors
                permuter(tab[i], tab[j])
fin
```

L'algorithmique : notions de base

16. Les fonctions

- Les fonctions sont des sous programmes indépendants qui permettent de réutiliser du code
- On définit une fonction par :
  - ✓ Son **identificateur** ou son **nom**.
  - ✓ Ses **arguments** ou **paramètres**, c'est à dire l'ensembles des valeurs dont elle a besoin pour fonctionner.
  - ✓ Son **type** de résultat, c'est à dire le type de valeur qui peux constituer le résultat de l'exécution de la fonction.
- Exemple : La fonction Somme (identificateur) nécessite deux valeurs entières qui seront stockées dans les paramètres a et b de la fonction. Elle fournit comme résultat un entier qui est la valeur de c avec l'instruction **retourne c** qui donne le résultat de la fonction.

```
fonction Somme (a : entier, b : entier) : entier
début
        entier c
        c ← a + b
        retourne c
fin
```

www.uha.fr

UNIVERSITÉ HAUTE-ALSACE

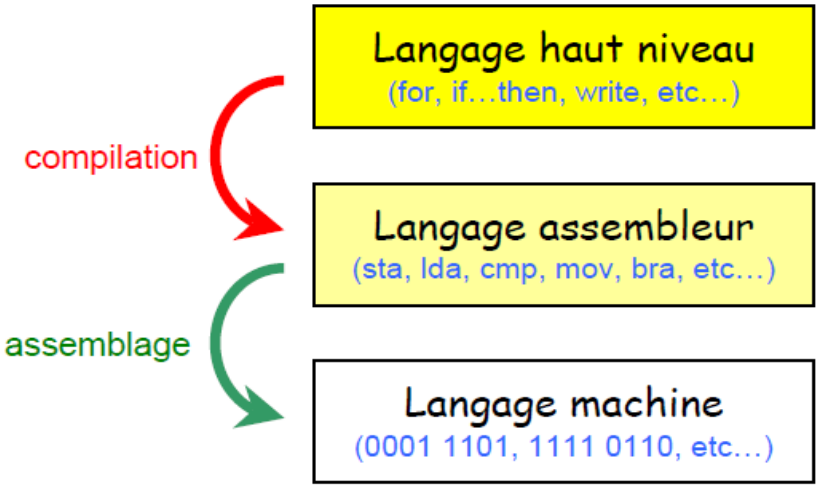
Langage de programmation

Langage de programmation

- Un **algorithme** exprime la structure logique d'un **programme** indépendamment du langage de programmation utilisé.
- La traduction de **l'algorithme** vers un **langage de programmation** particulier dépend du langage choisi et la mise en œuvre dépend de la plateforme d'exécution.
- La **programmation** consiste à « **expliquer** » en détail à l'ordinateur ce qu'il a à faire, sachant qu'il ne « comprend » pas le langage humain.
- Un programme n'est rien d'autre qu'une suite d'instructions, codées en **respectant scrupuleusement** un ensemble de **conventions fixées** à l'avance par un **langage de programmation**.
- La machine décode alors ces instructions en associant à chaque « **mot** » du langage de programmation une **action précise**.
- Pour se faire comprendre d'un ordinateur, il faudra donc faire appel à un **système de traduction** automatique, capable de convertir en nombres binaires les suites de caractères formant le code source d'un programme.

Langage de programmation

- Langage machine
  - ✓ langage compris par le microprocesseur.
- Langage assembleur
  - ✓ langage le plus « proche » du langage machine.
  - ✓ Il est composé par des instructions que l'on appelle des mnémoniques.
  - ✓ Chaque instruction représente un code machine différent.
- Langages de haut niveau
  - ✓ plus adaptés à l'être humain.
  - ✓ Permettent l'expression d'algorithmes sous une forme plus facile à apprendre (Python, C, Pascal, Java, etc…).
  - ✓ n'est donc pas compréhensible par le microprocesseur.



Langage de programmation

- Un ensemble de **mots-clés** et de **règles précises** indiquant comment agencer ces mots pour former des « phrases » que l'**interpréteur** ou le **compilateur** puisse traduire en langage machine.
- **Compilateur** : programme qui traduit un langage, le langage source, en un autre langage, le langage cible.
- **Interpréteur** : programme ayant pour tâche d'analyser et d'exécuter un programme écrit dans un langage source.
- Le **code source** subit alors une transformation (par le compilateur) ou une évaluation (pas l'interpréteur) dans une forme exploitable par la machine, ce qui permet d'obtenir un **programme**.
- La programmation est l'activité de **rédaction du code source** d'un programme, qui sera **analysé** puis **exécuté** par un ordinateur.

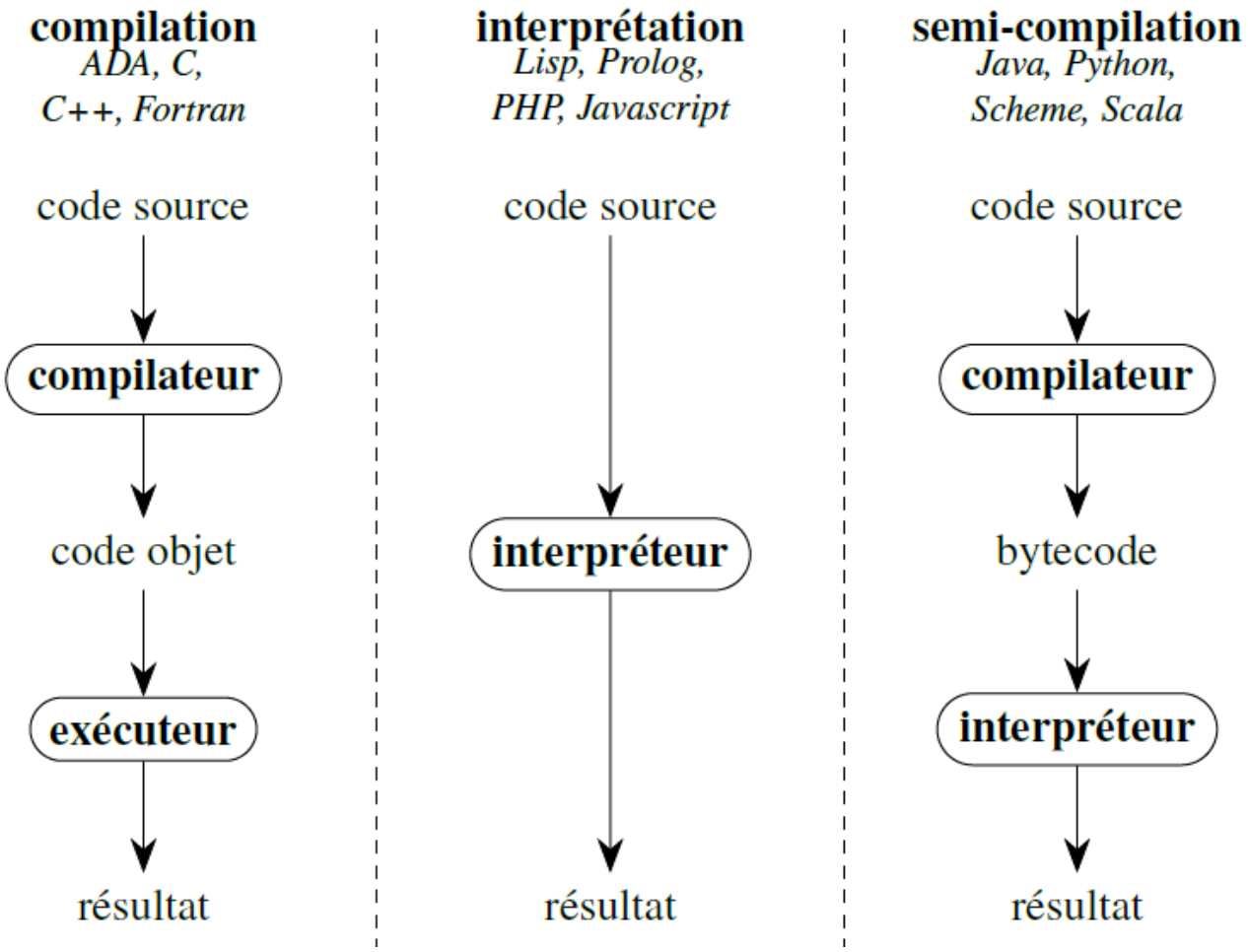
Langage de programmation

- **Compilation** : traduit la totalité du code source d'un programme en une seule fois vers une nouvelle suite de codes appelée **programme objet** (ou code objet). Celui-ci peut être **exécuté** indépendamment du compilateur et être conservé dans un fichier appelé fichier exécutable.

- **Interprétation** : traduit chaque ligne du programme source en quelques instructions en langage machine, directement exécutées au fur et à mesure. Aucun programme objet n'est généré. L'interpréteur doit être utilisé à chaque fois qu'on veut exécuter le programme.

- **Semi-compilation** : consiste à compiler le code source pour produire un code intermédiaire, appelé **bytecode,** similaire à un langage machine, mais pour une machine virtuelle. Ce bytecode sera transmis à un interpréteur à chaque exécution. C'est le cas de **Python** et **Java** par exemple.

Langage de programmation

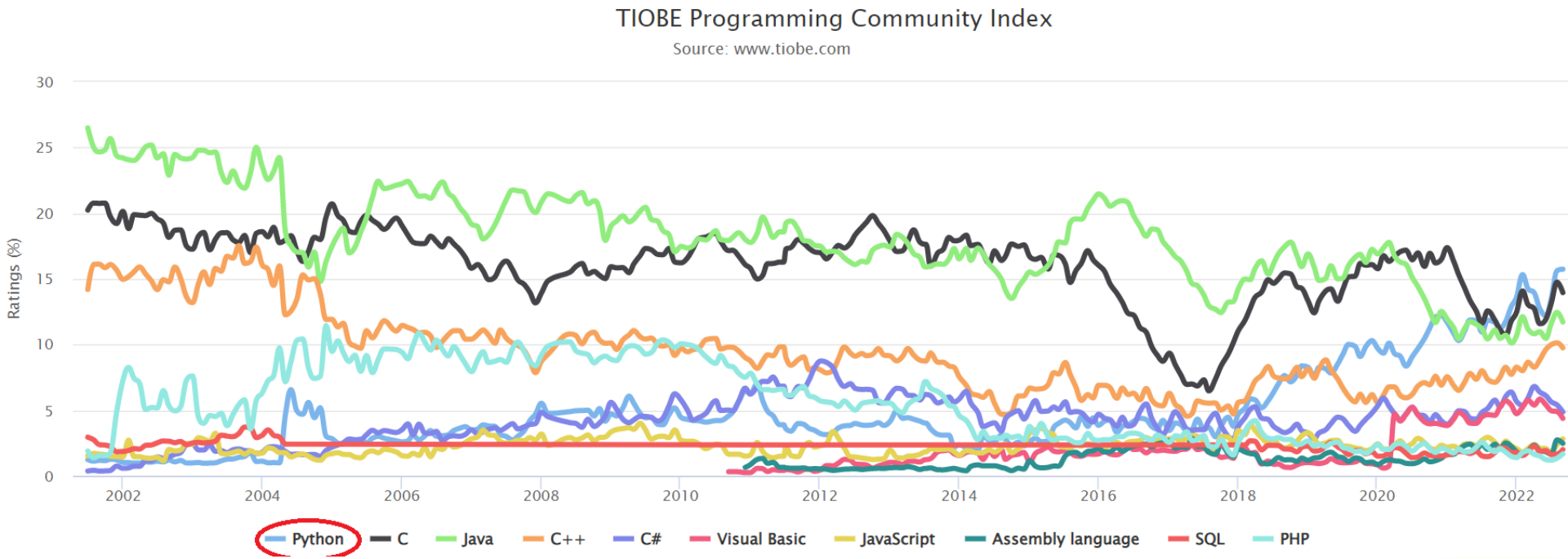


Langage de programmation

Langages :

- Nombreux langages différents.
- Différents paradigmes **: impératif, objet, fonctionnel**, etc.
- Différentes conventions de typage : **fort/faible, statique/dynamique, implicite/explicite.**
- Différentes conventions de priorité, de composition...
- Diverses spécialisations : système, web, jeux, applications graphiques...
- L'**algorithme** est indépendant du paradigme, il peut ensuite être traduit (implémenté) dans n'importe quel langage.

Langage de programmation



Quizz

- Comment participer ?
- Connectez-vous sur **www.wooclap.com/SXLQHB**

  Ou :