# Base d'Algorithmie topo :

# 1 Présentation

## 1.1Présentation informelle

Considérons les étapes qui interviennent dans la résolution problème quelconque :

1. concevoir une procédure qui une à fois appliquée amènera à une solution du problème ;
2. résoudre effectivement le problème en appliquant cette méthode.

Le résultat du premier point sera nommé un *algorithme*. Quant au deuxième point, c'est-à-dire la mise en pratique de l'algorithme, nous l'appellerons un *processus*.

Ces notions sont très répandues dans la vie courante. Un algorithme peut par exemple y prendre la forme :

- d'une recette de cuisine,
- d'un mode d'emploi,
- d'une notice de montage,
- d'une partition musicale,
- d'un texte de loi,
- d'un itinéraire routier.

Dans le cas particulier de l'informatique, une étape supplémentaire vient se glisser entre la conception de l'algorithme et sa réalisation à travers un processus : l'algorithme doit être rendu compréhensible par la machine que nous allons utiliser pour résoudre effectivement le problème. Le résultat de la traduction de l'algorithme dans un langage connu de la machine est appelé un *programme*.

## 1.2 Rapide historique

C'est un mathématicien perse du 8ème siècle, Al-Khawarizmi, qui a donné son nom à la notion d'algorithme. Son besoin était de traduire un livre de mathématiques venu d'Inde pour que les résultats et les méthodes exposés dans ce livre se répandent dans le monde arabe puis en Europe. Les résultats devaient donc être compréhensibles par tout autre mathématicien et les méthodes applicables, sans ambiguïté.

En particulier, ce livre utilisait une numérotation de position, ainsi que le chiffre zéro que ce type de représentation des nombres rend nécessaire. Par ailleurs, le titre de ce livre devait être repris pour désigner une branche des mathématiques, l'algèbre.

Si l'origine du mot *algorithme* est très ancienne, la notion même d'algorithme l'est plus encore : on la sait présente chez les babyloniens, 1800 ans avant JC.

Une tablette assyrienne provenant de la bibliothèque de Assourbanipal et datée de -640 expose une recette pour obtenir des pigments bleus (notons que pour avoir un algorithme satisfaisant, il faudrait préciser ici les temps de cuisson !) :

- tu ajouteras à un *mana* de bon *tersitu* un tiers de *mana* de verre *sirsu* broyé, un tiers de *mana* de sable, cinq *kisal* de craie,
- tu broieras encore,
- tu le recueilleras dans un moule, en le fermant avec un moule réplique,
- tu le placeras dans les ouvreaux du four,
- il rougeoira et *uknû merku* en sortira.

En résumé, il doit être bien clair que cette notion d'algorithme dépasse, de loin, l'informatique et les ordinateurs. Nécessite un vocabulaire partagé, des opérations de base maîtrisées par tous et de la précision.

# 2 Langage de description

## 2.1Premiers éléments

**Instructions de sorties**

On se donne une instruction *Écrire* pour afficher du texte, ainsi qu'une instruction *ALaLigne* pour pousuivre l'affichage à la ligne suivante.

**Patron d'un algorithme**

Le patron général est le suivant :

```
Algorithme NomAlgorithme
Début
  ... actions
Fin
```

La première ligne, appelée *profil* donne essentiellement le nom de l'algorithme. On trouve ensuite

un délimiteur de début puis les différentes actions composant l'algorithme et enfin un délimiteur de fin.

Ainsi, l'algorithme suivant est valide :

```
Algorithme Bonjour
Début
  Écrire('Hello world !!!')
  ALaLigne
Fin
```

Évoquer l'appel à un tel algorithme dans un *algorithme principal*.

## 2.2 Variables et types

Une variable est constitué d'un nom et d'un contenu, ce contenu étant d'un certain type. Les types différents : booléen, caractère, chaîne de caractères, nombre entier, nombre réel, etc.

Pour la clarté de l'algorithme, il peut être intéressant de déclarer les variables utilisées et leur type au tout début. Quand l'algorithme sera traduit en programme cette déclaration aura d'autres avantages : réservation de l'espace mémoire correspondant au type, possibilité de vérifier le programme du point de vue de la cohérence des types, etc.

Affectation souvent symbolisée par une flèche vers la gauche (←).

Expressions, évaluation, etc.

#### Nouveau patron d'un algorithme

Nous allons maintenant préciser les variables utilisées par l'algorithme et leurs types, en distingant les paramètres externes et les variables internes à l'algorithme. Ainsi, le patron d'un algorithme devient

```
Algorithme NomAlgorithme (paramètres...)
Variable ...
Début
  ... actions
Fin
```

## 2.3 Instructions conditionnelles et itératives

Le *Si Alors Sinon* va permettre de conditionner l'exécution de certaines instructions.

Le rôle des boucles est d'itérer un bloc d'instructions, soit un nombre précis de fois, soit relativement à une condition.

### Si Alors Sinon

```
Algorithme QueFaireCeSoir
Début
  Si Pluie
  Alors
    MangerPlateauTélé
    SeCoucherTot
  Sinon
    MangerAuRestaurant
    AllerAuCinema
  Fin si
Fin
```

### Boucle Fois

```
  Fois 3 faire
    Avancer
  Fin Fois
```

### Boucle *Tant que* et *Répéter*

```
Algorithme JusquAuMur
Début
  Tant que Non(DevantMur) faire
    Avancer
  Fin tant que
Fin

Algorithme JusquAuMurVersionRépéter
Début
  Répéter
    Avancer
  jusqu'à DevantMur
Fin
```

À noter que, contrairement à la boucle *tant que*, on passe toujours au moins une fois dans une boucle *répéter*. Ainsi, dans l'exemple ci-dessus, on avancera forcément d'une case... il conviendrait donc de tester si l'on n'est pas devant un mur avant d'utiliser cet algorithme.

**Boucle Pour**

Dotés des variables, nous pouvons maintenant écrire un nouveau type de boucle, la boucle *pour* :

```
Algorithme CompteJusqueCent
Début
  Pour i ← 1 à 100 faire
    Écrire(i)
    ALaLigne
  Fin Pour
Fin
```

Lorsque l'on sait exactement combien de fois on doit itérer un traitement, c'est l'utilisation de cette boucle qui doit être privilégiée.

Par exemple,

```
Algorithme DessineEtoiles (n : entier)
Variable i : entier
Début
  Pour i ← 1 à n faire
    Écrire('*')
  Fin pour
Fin
```

**Comparaisons des boucles pour un problème simple**

On reprend l'exemple précédent de la boucle *pour*

```
Algorithme CompteJusqueCentVersionPour
Variable i : entier
Début
  Pour i ← 1 à 100 faire
    Écrire(i)
    ALaLigne
  Fin Pour
Fin
```

et on écrit des algorithmes qui produisent la même sortie (les nombres de 1 à 100) mais en utilisant les différentes boucles.

D'abord, avec la boucle *tant que* (il faut initialiser *i* avant la boucle, et l'augmenter de 1 à chaque passage) :

```
Algorithme CompteJusqueCentVersionTQ
Variable i : entier
Début
  i ← 1
  Tant que (i ≤ 100) faire
    Écrire(i)
    ALaLigne
    i ← i+1
  Fin tant que
Fin
```

De même avec la boucle *répéter* (noter que la condition d'arrêt est ici la négation de la condition du *tant que*):

```
Algorithme CompteJusqueCentVersionRepeter
Variable i : entier
Début
  i ← 1
  Répéter
    Écrire(i)
    ALaLigne
    i ← i+1
  Jusqu'à (i > 100)
Fin
```

Enfin, en utilisant la récursivité, on obtient :

```
Algorithme CompteJusqueCentRecursif (n : entier)
Début
  Si (n ≤ 100)
  Alors
    Écrire(n)
    ALaLigne
    CompteJusqueCentRecursif(n+1)
  Fin si
Fin
```

# 3 Types abstraits

## 3.1 Formalisme

opérations et signature

constructeurs, accesseurs, modifieurs, testeurs, copieurs, afficheurs

pré-conditions et axiomes

opérations complexes

## 3.2 Le type abstrait « Entier »

**Type abstrait « Entier »**
Utilise « Booléen »

**Constructeurs**
zero : ∅ → Entier
succ : Entier → Entier
prec : Entier → Entier
*pré-condition pour prec(n) : estnul(n) est faux*

**Testeur**
estnul : Entier → Booléen

**Afficheur**
afficheentier : Entier → ∅

**Axiomes**
estnul(zero()) = vrai
estnul(succ(n)) = faux
succ(prec(n)) = n
prec(succ(n)) = n

Avec ce type abstrait, nous sommes capables de définir l'addition et la multiplication, indépendamment de l'implémentation de ce type.

```
Algorithme plus (n,m : entiers)
Début
    Tant que non(est_nul(n))
        m ← succ(m)
        n ← prec(n)
    Fin TQ
    Retourner m


Algorithme fois (n,m : entiers)
Début
    s ← zero()
    Tant que non(est_nul(n))
        s ← plus(s,m)
        n ← prec(n)
    Fin TQ
    Retourner s
Fin
```

## 3.3 Le type abstrait « Pile »

**Type abstrait « Pile »**
Utilise « Booléen » et « Élément »

**Constructeurs**
pilevide : ∅ → Pile
ajoute : Élément × Pile → Pile

**Accesseur**
sommet : Pile → Élément
*pré-condition pour sommet(p) : estvide(p) est faux*

**Modifieur**
depile : Pile → Pile
*pré-condition pour depile(p) : estvide(p) est faux*

**Testeur**
estvide : Pile → Booléen

**Afficheur**
affichepile : Pile → ∅

**Copieur**
copiepile : Pile → Pile

**Axiomes**
```
estvide(pilevide())  = vrai
estvide(ajoute(e,p)) = faux
sommet(ajoute(e,p))  = e
depile(ajoute(e,p))  = p
```

Un algorithme possible basé sur ce type :

```
Algorithme inversepile (p : Pile)
Début
    q ← copiepile(p)
    r ← pilevide()
    Tant que non(estvide(q))
        ajoute(sommet(q),r)
        depile(q)
    Fin TQ
    Retourner r
Fin
```

## 3.4 Type abstrait : le cas des types *produits*

Forme générale de ces types...

Sur un exemple : le type abstrait « Étudiant ».

**Type abstrait « Étudiant »**
Utilise « Entier » et « Texte »

**Constructeurs**
créer_étudiant : Texte × Texte × Texte → Étudiant

**Accesseurs**
```
nom_étudiant            : Étudiant         → Texte
prénom_étudiant         : Étudiant         → Texte
naissance_étudiant      : Étudiant         → Texte
noteinfo_étudiant       : Étudiant         → Entier
notemath_étudiant       : Étudiant         → Entier
notediscipline_étudiant : Texte × Étudiant → Entier
```

**Modifieur**
```
modifier_noteinfo : Étudiant × Entier → Étudiant
modifier_notemath : Étudiant × Entier → Étudiant
```

**Afficheur**
```
afficher_étudiant : Étudiant → ∅
```

**Axiomes**
```
nom_étudiant(créer_étudiant(n,p,d))        = n
prénom_étudiant(créer_étudiant(n,p,d))     = p
naissance_étudiant(créer_étudiant(n,p,d))  = d
noteinfo_étudiant(modifier_noteinfo(e,n))  = n
notemath_étudiant(modifier_notemath(e,n))  = n

notediscipline_étudiant('info',modifier_noteinfo(e,n)) = n
notediscipline_étudiant('math',modifier_notemath(e,n)) = n
```

## 3.4 Le type abstrait « Tableau »

Première version.

**Type abstrait « Tableau »**
Utilise « Entier » et « Élément »

**Constructeurs**
créer_tableau : Entier → Tableau

**Accesseurs**
contenu : Tableau × Entier → Élément
taille  : Tableau → Entier
*pré-condition pour contenu(t,n) : 1 ≤ n ≤ taille(t)*

**Modifieur**
fixecase :  Tableau × Entier × Élément → Tableau
*pré-condition pour fixecase(t,n,e) : 1 ≤ n ≤ taille(t)*

**Afficheur**
affichetableau : Tableau → ∅

**Copieur**
copietableau : Tableau → Tableau

**Axiomes**
taille(créer_tableau(n))   = n
contenu(fixecase(t,n,e),n) = e

Seconde version avec la notation *crochets* plus proche des langages de programmation.

**Type abstrait « Tableau »**
Utilise « Entier » et « Élément »

**Constructeurs**
créer_tableau : Entier → Tableau

**Accesseurs**
_[_] : Tableau × Entier → Élément
taille  : Tableau → Entier
*pré-condition pour contenu(t,n) : 1 ≤ n ≤ taille(t)*

**Modifieur**
_[_] = _ :  Tableau × Entier × Élément → Tableau
*pré-condition pour t[n]=e : 1 ≤ n ≤ taille(t)*

**Afficheur**
affichetableau : Tableau → ∅

**Copieur**
copietableau : Tableau → Tableau

**Axiomes**
taille(créer_tableau(n))   = n
contenu(fixecase(t,n,e),n) = e