Chapitre 6

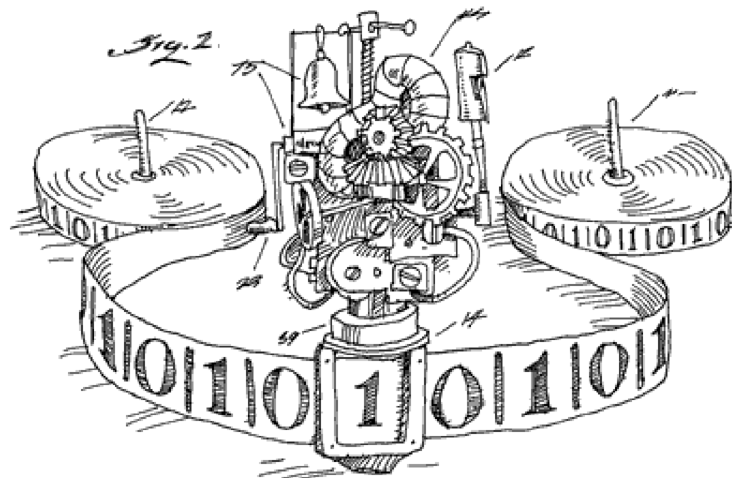
Programmation et langages

La programmation consiste à créer une séquence d'instructions pour un ordinateur afin qu'il puisse résoudre un problème ou exécuter une tâche.

6.1. La machine de Turing

Une **machine de Turing** est une machine théorique, inventée par Alan **Turing** en 1936, pour servir de modèle idéal lors d'un calcul mathématique. Ce modèle est toujours largement utilisé en informatique théorique, en particulier pour résoudre les problèmes de complexité algorithmique et de calculabilité.

Alan Mathison **Turing** (1912 - 1954)

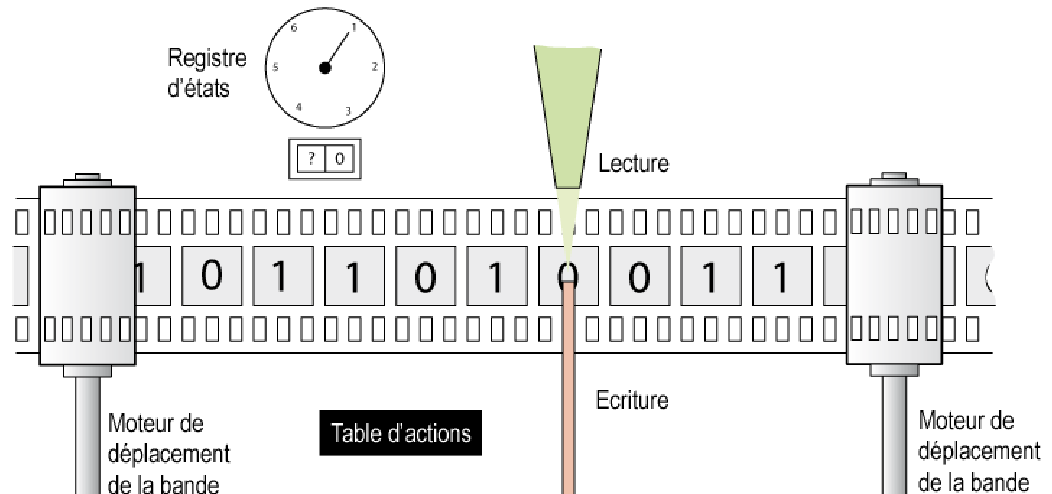


Turing Machine par Tom Dunne
American Scientist, Mars-Avril 2002

Une machine de Turing se compose des éléments suivants :

- Un « ruban » divisé en cases adjacentes. Chaque case contient un symbole parmi un alphabet fini. L'alphabet contient un symbole spécial « blanc » et un ou plusieurs autres symboles. Le ruban est de longueur infinie vers la gauche ou vers la droite (en d'autres termes, la machine doit toujours avoir assez de longueur de ruban pour son exécution). On considère que les cases non encore écrites du ruban contiennent le symbole « blanc ».
- Une « tête de lecture/écriture » qui peut lire et écrire les symboles sur le ruban, et se déplacer vers la gauche ou vers la droite du ruban.
- Un « registre d'état » qui mémorise l'état courant de la machine de Turing. Le nombre d'états possibles est toujours fini, et il existe un état spécial appelé « état de départ » qui est l'état initial de la machine avant son exécution.

- Une « table d'actions » qui indique à la machine quel symbole écrire, comment déplacer la tête de lecture (« G » pour une case vers la gauche, « D » pour une case vers la droite), et quel est le nouvel état, en fonction du symbole lu sur le ruban et de l'état courant de la machine. Si aucune action n'existe pour une combinaison donnée d'un symbole lu et d'un état courant, la machine s'arrête.



Les machines de Turing sont une abstraction des ordinateurs :

Voir le chapitre 2 pour se rappeler ce qu'est une mémoire, un bus et un microprocesseur.

- Le ruban représente la mémoire de l'ordinateur. Ceci comprend la mémoire centrale ainsi que les mémoires externes telles les disques durs. Contrairement à un ordinateur, la mémoire d'une machine de Turing est infinie.
- La tête de lecture/écriture représente le bus qui relie le microprocesseur à la mémoire. Une autre différence entre une machine de Turing et un ordinateur est que l'ordinateur peut accéder à la mémoire de manière directe, alors que la tête de lecture de la machine de Turing ne se déplace que d'une position à la fois.
- Le registre d'états et la table d'actions représentent le microprocesseur. Le nombre d'états est fini comme dans la réalité.

Exemple

La machine de Turing qui suit possède un alphabet {0, 1, blanc}. On suppose que le ruban contient une série de 1, et que la tête de lecture/écriture se trouve initialement au-dessus du 1 le plus à gauche. La table d'actions est la suivante :

État courant	Symbole lu	Symbole écrit	Mouvement	Nouvel état
e1	**0**	*(Arrêt)*		
	1	0	Droite	e2
e2	**1**	1	Droite	e2
	0 ou blanc	0	Droite	e3
e3	**1**	1	Droite	e3
	0 ou blanc	1	Gauche	e4
e4	**1**	1	Gauche	e4
	0 ou blanc	0	Gauche	e5
e5	**1**	1	Gauche	e5
	0 ou blanc	1	Droite	e1

Exécution :

La position de la tête de lecture/écriture est en jaune.

Étape	État	Ruban						
1	e1		**1**	1				
2	e2		0	**1**				
3	e2		0	1	**·**			
4	e3		0	1	0	**·**		
5	e4		0	1	**0**	1		
6	e5		0	**1**	0	1		
7	e5		**0**	1	0	1		
8	e1		1	**1**	0	1		
9	e2		1	0	**0**	1		
10	e3		1	0	0	**1**		
11	e3		1	0	0	1	**·**	
12	e4		1	0	0	**1**	1	
13	e4		1	0	**0**	1	1	
14	e5		1	**0**	0	1	1	
15	e1		1	1	**0**	1	1	
16		*(Arrêt)*						

Cette machine a pour effet de doubler le nombre de 1, en intercalant un 0 entre les deux séries. Par exemple, « 111 » deviendra « 1110111 ».

Exercice 6.1

Écrivez les tables d'actions des machines de Turing ci-dessous.
Pour les questions 2 à 4, on suppose, qu'au début, la tête de lecture/écriture est positionnée sur le symbole le plus à gauche.

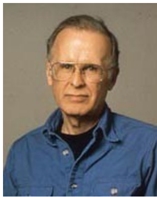
1. Une machine qui écrit 0 1 0 1 0 1 0 ... sur un ruban blanc.
2. Une machine qui multiplie par 2 le nombre entier positif écrit en binaire sur le ruban.
3. Une machine qui ajoute 1 au nombre entier positif écrit en binaire sur le ruban.
4. Une machine qui soustrait 1 au nombre entier positif écrit en binaire sur le ruban.

Konrad **Zuse** (1910 - 1995)

6.2. Un peu d'histoire

En 1936, la publication de l'article fondateur de la science informatique *On Computable Numbers with an Application to the Entscheidungsproblem*, par Alan Mathison **Turing**, allait donner le coup d'envoi à la création de l'ordinateur programmable. Il y présente sa **machine de Turing**, le premier calculateur universel programmable, et invente les concepts et les termes de programmation et de programme.

En 1948, Konrad **Zuse** publie un article sur son langage de programmation qu'il a développé entre 1943 et 1945 : le **Plankalkül**. Zuse le considère comme étant le premier langage de haut



John **Backus**
(1924 - 2007)

Guido van **Rossum**
(né en 1956)

niveau.

C'est à partir des années 50 que l'on verra apparaître les premiers langages de programmation modernes. Voici les créateurs des langages les plus utilisés :

- John **Backus**, inventeur de Fortran (1954)
- John **McCarthy**, inventeur de LISP (1958)
- Grace **Hopper**, surnommée « la mère du langage COBOL » (1959)
- John George **Kemeny**, concepteur du BASIC (1963)
- Dennis **Ritchie** et Ken **Thompson**, inventeurs du langage C (1972)
- Niklaus **Wirth** inventeur de Pascal (1970) et Modula-2 (1977)
- Bjarne **Stroustrup**, développeur de C++ (1985)
- Guido **van Rossum**, créateur de Python (1991)
- James **Gosling** et Patrick **Naughton**, créateurs de Java (1991).

Il existe des centaines de langages de programmation, certains disent des milliers. Les plus populaires en 2025 sont, dans l'ordre, Python, C++, Java, C, C#, Javascript, SQL, Go, Delphi, Visual Basic, Fortran, Scratch, Rust, PHP, R, ...[1]

6.2.1. Évolution des langages informatiques

On distingue aujourd'hui cinq générations de langages.

La première génération est le langage machine, ou code machine. On parle aussi de langage natif. Il est composé d'instructions et de données à traiter codées en binaire. C'est le seul langage qu'un ordinateur peut traiter directement.

Voici à quoi peut ressembler un programme en langage machine :

```
A1 01 10 03 06 01 12 A3 01 14
```

Il s'agit de la représentation hexadécimale d'un programme permettant d'additionner les valeurs de deux cases mémoire et de stocker le résultat dans une troisième case. On voit immédiatement la difficulté d'un tel langage...

La deuxième génération est le langage assembleur[2] : le code devient lisible et compréhensible par un plus grand nombre d'initiés. Il existe en fait un langage assembleur par type de processeur.

Le programme précédent écrit en assembleur donnerait ceci :

```
MOV AX, [0110]
ADD AX, [0112]
MOV [0114], AX
```

Il reste utilisé dans le cadre d'optimisations, mais a été supplanté en popularité par les langages plus accessibles de troisième génération.

La troisième génération utilise une syntaxe proche de l'anglais. Proposés autour de 1960, ces langages ont permis un gain énorme en lisibilité et en productivité. Ils ne dépendent plus du processeur, comme c'était le cas des générations précédentes, mais d'un *compilateur*[3] spécifique du processeur. L'idée de *portabilité*[4] des programmes était lancée.

La plupart des langages de programmation actuels sont de troisième génération. On trouve dans cette catégorie tous les grands langages : **Ada**, **Algol**, **Basic, Cobol**, **Eiffel**, **Fortran**, **C**, **C++**, **Java**, **Perl**, **Pascal**, **Python**, **Ruby**, ... Cette génération couvre d'ailleurs tant de langages qu'elle est souvent subdivisée en catégories, selon le paradigme[5] particulier des langages.

Les langages de **quatrième génération**, abrégés L4G, souvent associée à des bases de données,

1 Source : https://www.tiobe.com/tiobe-index/
2 Pour s'initier à l'assembleur : www.commentcamarche.net/contents/asm/assembleur.php3
3 En pratique, un **compilateur** sert le plus souvent à traduire un code source écrit dans un langage de programmation en un autre langage, habituellement un langage assembleur ou un langage machine. Le premier compilateur, A-0 System, a été écrit en 1951 par Grace **Hopper**.
4 En informatique, la **portabilité** d'un programme est caractérisée par sa capacité à fonctionner plus ou moins facilement dans différents environnements d'exécution
5 Voir paragraphe 6.4

se situent un niveau au-dessus, en intégrant la gestion de l'interface utilisateur[6] et en proposant un langage moins technique, plus proche de la syntaxe naturelle.

Ils sont conçus pour un travail spécifique : gestion de base de données (**Microsoft Access**, **SQL**), production graphique (**Postscript**), création d'interface (**4D**).

La cinquième génération de langages sont des langages destinés à résoudre des problèmes à l'aide de contraintes, et non d'algorithmes écrits. Ces langages reposent beaucoup sur la logique et sont particulièrement utilisés en intelligence artificielle. Parmi les plus connus, on trouve **Prolog**, dont voici un exemple :

```
frère_ou_soeur(X,Y) :- parent(Z,X), parent(Z,Y), X \= Y.
parent(X,Y) :- père(X,Y).
parent(X,Y) :- mère(X,Y).
mère(trude, sally).
père(tom, sally).
père(tom, erica).
père(mike, tom).
```

Il en résulte que la demande suivante est évaluée comme vraie :

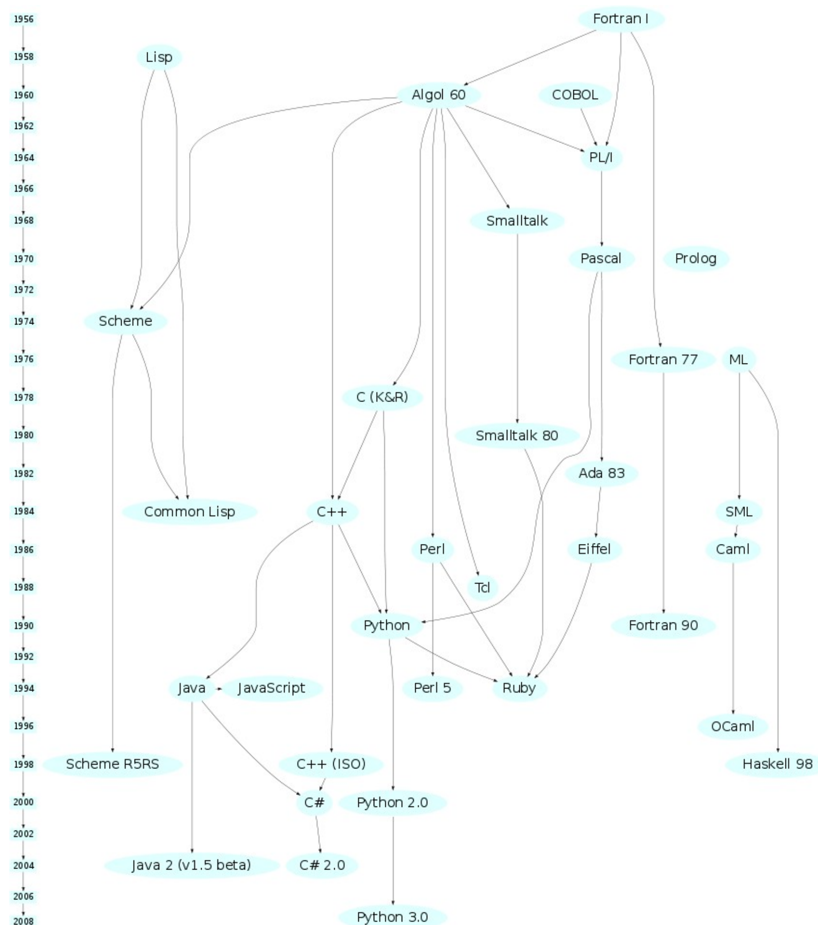
```
?- frère_ou_soeur(sally, erica).
     oui.
```

Ce qui signifie que Sally et Erica sont sœurs. En effet, Sally et Erica ont le même père (Tom).

6.2.2. Quelques langages courants

Selon certaines sources, il existe environ 2500 langages de programmation, certains étant très généraux (**C**, **Java**), d'autres hyper-spécialisés. Par exemple, **RobotProg** permet de déplacer un robot virtuel, **R** est un langage pour la statistique, etc.

Par comparaison, une majorité de spécialistes s'accordent sur le chiffre d'environ 6000 langues humaines parlées et/ou écrites de par le monde.



6 L'**interface utilisateur** définit les moyens et outils mis en œuvre pour qu'un humain puisse contrôler et communiquer avec une machine.

6.2.3. « Hello world ! »

Le nom **Ada** a été choisi en l'honneur d'Ada **Lovelace** (1815-1852), qui est supposée avoir écrit le premier programme de l'histoire.
La première version d'**Ada** remonte à 1983.

C'est dans un memorandum interne de Brian **Kernighan**, *Programming in C : A tutorial*, écrit en 1974 dans les laboratoires *Bell*, que l'on trouve la première version d'un mini-programme affichant à l'écran « Hello World! ». Voici comment cela s'écrit dans divers langages[7] :

Ada

```
with Ada.Text_IO;
 use Ada.Text_IO;

 procedure Bonjour is
 begin -- Bonjour
   Put("Hello world!");
 end Bonjour;
```

Assembleur X86 sous DOS

```
cseg segment
 assume cs:cseg, ds:cseg
 org 100h
 main proc
 jmp debut
 mess db 'Hello world!$'
 debut:
 mov dx, offset mess
 mov ah, 9
 int 21h
 ret
 main endp
 cseg ends
 end main
```

BASIC est un acronyme pour Beginner's All-purpose Symbolic Instruction Code. Il a été conçu à la base en 1963 par John George **Kemeny** et Thomas Eugene **Kurtz**.

BASIC

```
10 PRINT "Hello world!"
20 END
```

Inventé au début des années 1970 avec UNIX, **C** est devenu un des langages les plus utilisés.

C

```
#include <stdio.h>

int main()/* ou int argc, char *argv[] */
{
    printf("Hello world!\n");
    return 0;
}
```

Bjarne **Stroustrup** a développé **C++** au cours des années 1980. Il s'agissait d'améliorer le langage **C**.

C++

```
#include <iostream>

int main()
{
        std::cout << "Hello world!" << std::endl;
        return 0;
}
```

Fortran (FORmula TRANslator) est utilisé dans les applications de calcul scientifique.

FORTRAN 77

```
PROGRAM BONJOUR
WRITE (*,*) 'Hello world!'
END
```

7 Une liste plus complète est disponible sur Wikipédia : http://fr.wikipedia.org/wiki/Hello_world

Java a été créé par Sun Microsystems, et présenté officiellement le 23 mai 1995.

JavaScript est un langage de programmation de scripts utilisé dans les pages web interactives.

Java

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

Javascript

```
document.write("Hello world!");
```

Python 1 et 2

```
print "Hello world!"
```

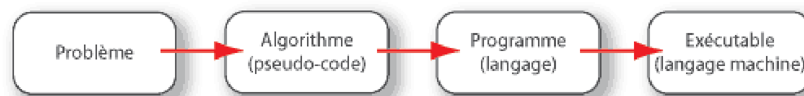
Python 3

```
print("Hello world!")
```

6.3. Transformation du code source

Le code source n'est (presque) jamais utilisable tel quel. Il est généralement écrit dans un langage « de haut niveau », compréhensible pour l'homme, mais pas pour la machine. Il existe deux stratégies de traduction, ces deux stratégies étant parfois disponibles au sein du même langage.

- Le langage traduit les instructions au fur et à mesure qu'elles se présentent. Cela s'appelle la compilation à la volée, ou l'interprétation.
- Le langage commence par traduire l'ensemble du programme en langage machine, constituant ainsi un deuxième programme (un deuxième fichier) distinct physiquement et logiquement du premier. Ensuite, et ensuite seulement, il exécute ce second programme. Cela s'appelle la compilation.



6.3.1. Compilation

Certains langages sont « compilés ». En toute généralité, la compilation est l'opération qui consiste à transformer un langage source en un langage cible. Dans le cas d'un programme, le compilateur va transformer tout le texte représentant le code source du programme, en code compréhensible pour la machine, appelé code machine.

Dans le cas de langages compilés, ce qui est exécuté est le résultat de la compilation. Une fois effectuée, l'exécutable obtenu peut être utilisé sans le code source.

6.3.2. Interprétation

D'autres langages ne nécessitent pas de phase spéciale de compilation. La méthode employée pour exécuter le programme est alors différente. Le programme entier n'est jamais compilé. Chaque ligne de code est compilée « en temps réel » par un programme. On dit de ce programme qu'il interprète le code source. Par exemple, Python est un langage interprété.

Cependant, ce serait faux de dire que la compilation n'intervient pas. L'interprète produit le code machine, au fur et à mesure de l'exécution du programme, en compilant chaque ligne du code source.

6.3.3. Avantages, inconvénients

Les avantages généralement retenus pour l'utilisation de langages « compilés », est qu'ils sont

plus rapides à l'exécution que des langages interprétés, car l'interprète doit être lancé à chaque exécution du programme, ce qui mobilise systématiquement les ressources.

Les langages interprétés offrent en revanche une certaine portabilité, ainsi qu'une facilité pour l'écriture du code. En effet, il n'est pas nécessaire de passer par la phase de compilation pour tester le code source.

6.4. Paradigmes

En informatique, un **paradigme** est une façon de programmer, un modèle qui oriente notre manière de penser pour formuler et résoudre un problème.

Certains langages sont conçus pour supporter un paradigme en particulier (**Smalltalk** et **Java** supportent la programmation orientée objet, tandis que **Haskell** est conçu pour la programmation fonctionnelle), alors que d'autres supportent des paradigmes multiples (à l'image de **C++**, **Common Lisp**, **OCaml**, **Python**, **Ruby** ou **Scheme**).

6.4.1. Programmation impérative

C'est le paradigme le plus ancien. Les recettes de cuisine et les itinéraires routiers sont deux exemples familiers qui s'apparentent à de la programmation impérative. La grande majorité des langages de programmation sont impératifs. Les opérations sont décrites en termes de séquences d'instructions exécutées par l'ordinateur pour modifier l'état du programme.

6.4.2. Programmation structurée (ou procédurale)

Niklaus **Wirth**
(1934 - 2024)

Edsger Wybe
Dijkstra
(1930 - 2002)

La programmation structurée constitue un sous-ensemble de la programmation impérative. C'est un paradigme important de la programmation, apparu vers 1970. Elle dérive de travaux de Niklaus **Wirth** pour son **Algol W** et reçut son coup d'envoi avec l'article fondateur de **Dijkstra** dans Communications of the ACM intitulé *GO TO statement considered harmful*.

Elle est en effet célèbre pour son essai de suppression de l'instruction GOTO ou du moins pour la limitation de son usage.

La programmation structurée est possible dans n'importe quel langage de programmation procédural, mais certains, comme le **Fortran IV**, s'y prêtaient très mal. Vers 1970, la programmation structurée devint une technique populaire, et les langages de programmation procéduraux intégrèrent des mécanismes rendant aisée la programmation structurée. Parmi les langages de programmation les plus structurants, on trouve **PL/I**, **Pascal** et, plus tardivement, **Ada**.

Exemple

Dans certains langages anciens comme le **BASIC**, les lignes de programmation portent des numéros, et les lignes sont exécutées par la machine dans l'ordre de ces numéros. Dans tous ces langages, il existe une instruction de branchement qui envoie directement le programme à la ligne spécifiée (GOTO).

Prenons l'exemple d'une structure « **Si … Alors … Sinon** » :

<table>
<tr><th>Programmation Structurée</th><th>Programmation non structurée</th></tr>
<tr><td><pre>Si condition Alors
    instructions 1
Sinon
    instructions 2
FinSi</pre></td><td><pre>1000 Si condition Alors Aller En 1200
1100 instruction 2
1110 etc.
1120 etc.
1190 GOTO 1400
1200 instruction 1
1210 etc.
1220 etc.
1400 suite de l'algorithme</pre></td></tr>
</table>

Les programmeurs décomposent leur code en **procédures** ne dépassant guère 50 lignes, afin d'avoir le programme en entier sous leurs yeux.

Une procédure, aussi appelée routine, sous-routine, module ou fonction, contient une série d'étapes à réaliser. N'importe quelle procédure peut être appelée à n'importe quelle étape de l'exécution du programme, incluant d'autres procédures, voire la procédure elle-même (récursivité).

Il n'y a que des avantages à découper un programme en procédures :

- on a la possibilité de réutiliser le même code à différents emplacements dans le programme sans avoir à le retaper ;
- il est plus simple de suivre l'évolution du programme (la programmation procédurale permet de se passer d'instructions « GOTO ») ;
- on crée un code plus modulaire et structuré ;
- chaque programmeur peut développer son bout de code de son côté.

6.4.3. Programmation orientée objet

La **programmation orientée objet** (POO) ou programmation par objet, est un paradigme de programmation informatique qui consiste en la définition et l'assemblage de « briques logicielles » appelées objets. Un objet représente un concept, une idée ou toute entité du monde physique, comme une voiture, une personne ou encore une page d'un livre. Le langage **Simula-67** jette les prémisses de la programmation objet, résultat des travaux sur la mise au point de langages de simulation informatique dans les années 1960 dont s'inspira aussi la recherche sur l'intelligence artificielle dans les années 1970-80. Mais c'est réellement par et avec **Smalltalk 72** puis **Smalltalk 80**, inspiré en partie par **Simula**, que la programmation par objets débute et que sont posés les concepts de base de celle-ci : objet, messages, encapsulation, polymorphisme, héritage, etc.

À partir des années 1980, commence l'effervescence des langages à objets : **Objective C** (début des années 1980), **C++** (C with classes) en 1983, **Eiffel** en 1984, **Common Lisp Object System** dans les années 1980, etc. Les années 1990 voient l'âge d'or de l'extension de la programmation par objet dans les différents secteurs du développement logiciel.

Quelques langages à objets : **Ada**, **Java**, **C++**, **C#**, **Objective C**, **Eiffel**, **Python**, **PHP**, **Smalltalk**, ...

6.4.4. Programmation fonctionnelle

La programmation fonctionnelle est un paradigme de programmation qui considère le calcul en tant qu'évaluation de fonctions mathématiques et rejette le changement d'état et la mutation des données. Elle souligne l'application des fonctions, contrairement au modèle de programmation impérative qui met en avant les changements d'état.

Le langage fonctionnel le plus ancien est **Lisp**, créé en 1958 par McCarthy. **Lisp** a donné naissance à des variantes telles que **Scheme** (1975) et **Common Lisp** (1984). **Haskell** (1987) est aussi un langage à paradigme fonctionnel. La programmation fonctionnelle s'affranchit de façon radicale des effets secondaires en interdisant toute opération d'affectation.

Le paradigme fonctionnel n'utilise pas de machine d'états pour décrire un programme, mais un emboîtement de « boîtes noires » que l'on peut imbriquer les unes dans les autres. Chaque boîte possédant plusieurs paramètres en entrée mais une seule sortie, elle ne peut sortir qu'une seule valeur possible pour chaque n-uplet de valeurs présentées en entrée. Ainsi, les fonctions n'introduisent pas d'effets de bord. Un programme est donc une application, au sens mathématique, qui ne donne qu'un seul résultat pour chaque ensemble de valeurs en entrée. Cette façon de penser, qui est très différente de la pensée habituelle en programmation impérative, est l'une des causes principales de la difficulté qu'ont les programmeurs formés aux langages impératifs pour aborder la programmation fonctionnelle. Cependant, elle ne pose généralement pas de difficultés particulières aux débutants qui n'ont jamais été exposés à des langages impératifs.

Exercice 6.2

Expliquez cette description de **Python**, tirée de Wikipédia :

« **Python** est un langage de programmation interprété multi-paradigme. Il favorise la programmation impérative structurée, et orientée objet. »

6.5. Pseudo-code

En programmation, le pseudo-code est une façon de décrire un algorithme en respectant certaines conventions, mais sans référence à un langage de programmation en particulier. L'écriture en pseudo-code permet de développer une démarche structurée, en « oubliant » temporairement la syntaxe rigide d'un langage de programmation.

6.5.1. Conventions

Il n'existe pas de convention universelle pour le pseudo-code. En voici une.

Nom de l'algorithme

Par exemple : Calcul de la date de Pâques

Ces trois informations (nom, données et résultat) seront données en préambule.

Données d'entrées

Par exemple : Année > 1582

Résultat en sortie

Par exemple : Date de Pâques pour l'année donnée

Tableaux

On peut voir un tableau comme un meuble de rangement avec des tiroirs numérotés.

Tableau unidimensionnel (ou liste)

`A[i]` : l'élément de rang `i` dans le tableau `A`. Souvent, les cellules du tableau sont numérotées à partir de 0, mais certains langages (Pascal, Scratch, ...) les numérotent à partir de 1.

Tableau bidimensionnel `A[i,j]`

Élément de la ligne `i` et de la colonne `j` du tableau `A`.

Les guillemets sont importants !

Chaîne de caractères

`"Hello World !"`

Boucles

Il y a 3 types de boucles.

Il est possible qu'une boucle de type **tant que** ne s'exécute pas, alors qu'une boucle de type **répéter** s'exécutera au moins une fois. Quant aux boucles de type **pour**, on sait par avance combien de fois elles s'exécuteront.

Bloc tant que

```
TANT QUE condition FAIRE
    instruction
    ...
FIN TANT QUE
```

Bloc répéter jusqu'à

```
REPETER
    instruction
    ...
JUSQU'A condition
```

Bloc pour

```
POUR i ALLANT DE d À f FAIRE
    instruction
    ...
FIN POUR
```

Condition

Bloc si

```
SI condition ALORS
    instruction-si-vrai
    ...
SINON
    instruction-si-faux
    ...
FIN SI
```

On met la valeur 10 dans la variable n

Affectation

```
n ← 10   (ou n:=10)
```

Fonction

Sous-programmes

En informatique, « retourner » signifie « envoyer ».

Les fonctions retournent un ou plusieurs résultats, contrairement aux procédures.

```
FONCTION nom(paramètres)
    instruction
    ...
    RETOURNER résultat
```

Procédure

```
PROCEDURE nom(paramètres)
    instruction
    ...
```

Commentaires

Pour aider à la lecture et la compréhension.

```
/* ceci est un commentaire */
```

6.5.2. En pratique

- **Règle d'or :** le pseudo-code doit être suffisamment précis pour que quelqu'un d'autre l'ayant lu sans connaître l'algorithme soit capable de l'appliquer et éventuellement de le coder sur un ordinateur.
- Le pseudo-code d'un petit algorithme prend en général une douzaine de lignes. Pour un problème plus complexe, il faut compter jusqu'à 20 ou 25 lignes.
- Prévoyez de faire tenir tout le pseudo-code sur une seule page.
- Pour améliorer la lisibilité, indentez vers la droite le corps des boucles et des fonctions.
- Lorsqu'on écrit le brouillon du pseudo-code, il est judicieux de laisser des lignes blanches régulièrement pour pouvoir ajouter des choses auxquelles on n'avait pas immédiatement pensé.

6.5.3. Exemple de pseudo-code

Euclide
(~300 av. J.-C.)

Appliquez cet algorithme avec $A = 136$ et $B = 96$.

Algorithme : Algorithme d'Euclide
Données : Deux nombres entiers A et B (avec $A > B$)
Résultat : PGDC de A et B

```
REPETER
    C ← A modulo B            /* reste de division entière */
    SI C ≠ 0 ALORS
        A ← B
        B ← C
    FIN SI
JUSQU'A CE QUE C = 0
écrire B                      /* B est le PGDC */
```

6.6. Organigrammes de programmation

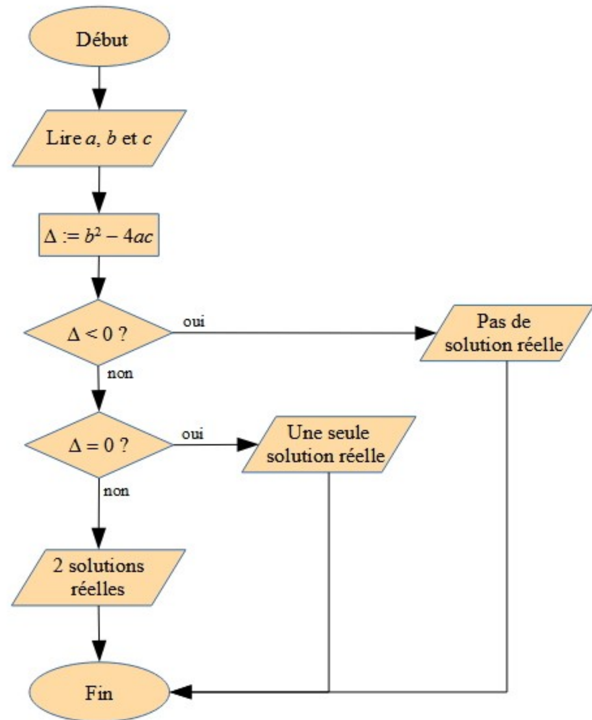
La rédaction d'un algorithme sous forme de texte n'est pas toujours la plus claire. On peut aussi présenter un algorithme sous une forme plus visuelle : un organigramme. On parle aussi de logigramme, d'algorigramme, ou, plus rarement, d'ordinogramme (*flowchart* en anglais).

Plusieurs symboles seront utilisés (rectangles, losanges, parallélogrammes, ovales), avec ces significations :

- Un ovale indique le début et la fin du programme.
- Un parallélogramme désigne une entrée (lire une donnée) ou une sortie (écrire un résultat).
- Un rectangle désigne une modification des variables.
- Un losange est un test conditionnel qui débouche sur deux possibilités (oui/non).

Ces formes suivent la norme ISO 5807:1985, qui décrit en détail les différents symboles à utiliser pour représenter un programme informatique de manière normalisée.
Il vous en coûtera 118 CHF pour la consulter...

L'organigramme ci-contre permet de déterminer le nombre de solutions réelles d'une équation du second degré donnée sous la forme $ax^2 + bx + c = 0$.



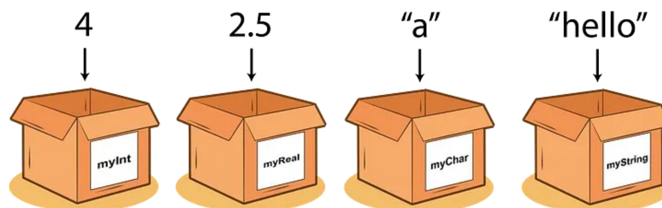
Exercice 6.3

Représentez sous forme d'organigramme l'algorithme d'Euclide décrit au § 6.5.3.

6.7. Les notions principales de la programmation

6.7.1. Variables et affectation

Une affectation est une opération qui permet d'attribuer une valeur à une variable. Une variable est une sorte de boîte, qui a un nom, et qui contient une valeur : un nombre, une lettre, un mot, etc.



L'instruction `x←t` (aussi notée `x:=t`) consiste à remplir la boîte `x` avec la valeur de l'expression `t`. Si `x` contenait déjà une valeur, celle-ci est écrasée.

Exemple

Après l'affectation `x ← 3`, la variable `x` contiendra la valeur 3 (initialisation).
Après l'affectation `x ← x+2`, la variable `x` contiendra la valeur 5 (3+2).
Après l'affectation `x ← x*x`, la variable `x` contiendra la valeur 25 (5·5).

Affectation et comparaison

Il ne faut pas confondre l'*affectation* avec la *comparaison*. Par exemple, la comparaison `x==3` permet de savoir si oui ou non la valeur de la variable `x` est 3. Dans la plupart des langages, on différencie ces deux opérations. En **Python**, l'affectation est exprimée par un « = », tandis que la comparaison est exprimée par « == ». D'autres langages (**Pascal** par exemple) utilisent respectivement les opérateurs « **:** = » et « = ».

Incrémentation / décrémentation

Il arrive fréquemment que l'on doive *incrémenter* (ou *décrémenter*) une variable, c'est-à-dire augmenter (ou diminuer) sa valeur d'une ou plusieurs unités.

Dans ce cas, on peut utiliser l'instruction `x` ← `x+1`. Voici ce qui se passe : on prend la valeur contenue dans `x`, on y ajoute 1, puis on remet le nouveau résultat dans la variable `x`, à la place de l'ancienne valeur, qui est donc écrasée.

L'instruction `x` ← `x+1` est tellement fréquente qu'il existe souvent des raccourcis : `x += 1` (**Python**) ou encore `x++` (**C, Mathematica**).

6.7.2. Opérateurs logiques

Les opérateurs logiques sont des symboles ou des mots-clés utilisés pour combiner des expressions et obtenir un résultat booléen (vrai ou faux). Les plus courants sont ET, OU et NON. Ils permettent de créer des conditions complexes :

- **ET (AND)** : l'expression est vraie uniquement si les deux conditions sont vraies.
- **OU (OR)** : l'expression est vraie si au moins une des conditions est vraie.
- **NON (NOT)** : l'expression est vraie si la condition est fausse, et fausse si elle est vraie.

L'ordre de priorité, du plus élevé au plus bas est NON, ET, OU. On peut, comme en maths, utiliser des parenthèses pour changer les priorités et/ou rendre l'écriture plus claire.

Exercice 6.4

Remarque préliminaire

Pour savoir si un nombre naturel n est divisible par un nombre naturel d, on calcule le reste de la division euclidienne, appelé « modulo », et qui se note $n \% d$. Par exemple, $18 \% 7 = 4$, puisque $18 \div 7 = 2$ reste 4. 18 n'est donc pas divisible par 7, car le reste n'est pas nul.

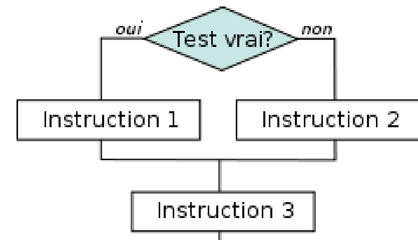
Soit $a = 5$ et $b = 25$.
Remplissez la colonne « Résultat » du tableau ci-dessous avec VRAI ou FAUX.

	Condition	Résultat
1.	$a < b$	vrai
2.	$-a \geq -b$	
3.	$b \% a = 0$	
4.	$(a < 10)$ ou $(b < 10)$	
5.	$(a < 10)$ et $(b < 10)$	
6.	$(a < 10)$ et non $(b < 10)$ ou $(a+b < 50)$	
7.	$(a+b=30)$ et non $(a \cdot b > 100)$	
8.	$(a > 4)$ et non $(b \% 10 = 5)$	
9.	$((a = 5)$ ou $(b < 10))$ et non $(a + b > 20)$	
10.	non $(a > 2$ et $b < 30)$ ou $(a + b = 40)$	
11.	$(b/a = 5)$ et (non $(a \cdot 2 = 10)$ ou $(b - 5 > 10)$)	

6.7.3. Les tests

Un test est une instruction du genre **si… alors… sinon**. La suite du programme dépendra du résultat du test.

```
SI Test vrai ALORS
   Instruction 1
SINON
   Instruction 2
FIN SI
Instruction 3
```

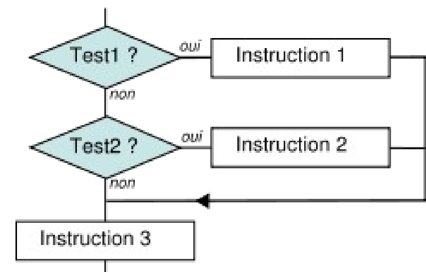


Par exemple, en **Python**, on aura :

```
if x>=10:
    x=x-20
else:
    x=x+2
print(x)
```

On peut enchaîner autant d'instructions « *sinon si* » que l'on veut : seule la première dont la condition sera vérifiée sera exécutée. On peut généralement associer une clause *sinon* qui ne sera exécutée que si aucune clause *sinon si* n'a été vérifiée.

```
SI Test1 vrai  ALORS
   Instruction 1
SINON SI Test2 vrai ALORS
   Instruction 2
FIN SI
Instruction 3
```



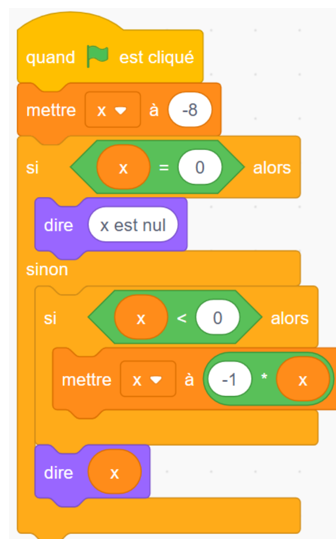
Par exemple, en **Python 3** :

== : comparaison

= : affectation

```
if x==0:
    print("x est nul")
elif x<0:
    x = -x
print("valeur absolue =",x)
```

En en **Scratch** :



Exercice 6.5

Depuis l'adoption du calendrier grégorien en 1582, une année est bissextile (elle a 366 jours) dans l'un des deux cas suivants :

1. si l'année est divisible par 4 et non divisible par 100 ;
2. si l'année est divisible par 400.

Sinon, l'année n'est pas bissextile : elle a la durée habituelle de 365 jours.

Dessinez un organigramme permettant de déterminer si une année est bissextile ou commune (sans utiliser les mots ET/OU dans les tests conditionnels).

Exercice 6.6

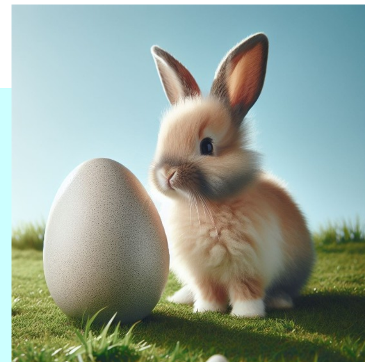
Selon la définition établie par le Concile de Nicée en 325 :

« Pâques est le dimanche qui suit le 14ème jour de la Lune qui atteint cet âge le 21 mars ou immédiatement après. »

Autrement dit, Pâques est le dimanche qui suit la pleine lune de printemps (c'est-à-dire la première pleine lune qui se produit le 21 mars ou après). Selon cette définition, Pâques tombe entre le 22 mars et le 25 avril.

Il existe plusieurs algorithmes assez complexes pour calculer cette date. Voici une version simplifiée de l'agorithme de Gauss, qui fonctionne seulement entre 1901 et 2099 :

Algorithme : Algorithme de Gauss (simplifié)
Données : Une année entre 1901 et 2099
Résultat : La date de Pâques



Rappel :
A%4 (A modulo 4) est le reste de la division entière de A par 4.

```
Lire l'année A
R ← A%4
S ← A%7
T ← A%19
B ← 19·T + 24
M ← B%30
C ← 2·R + 4·S + 6·M + 5
N ← C%7
P ← M + N
SI P ≤ 9 ALORS
   écrire P+22 "mars"
SINON
   écrire P-9 "avril"
FIN SI
```

Quelle est la date de Pâques cette année ?

Programmez cet algorithme dans le langage de votre choix (Scratch, Python, …).

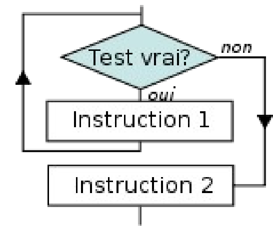
6.7.4. Les boucles

Une boucle est une structure de contrôle permettant de répéter une ou un ensemble d'instructions plusieurs fois, tant qu'une condition est satisfaite.

Quelle est la différence des boucles utilisées par bip-bip et le Coyote ?

Boucle « Tant que »

```
TANT QUE Test vrai FAIRE
     Instruction 1
FIN TANT QUE
Instruction 2
```



Par exemple, en **Python 3**, ce programme écrira tous les entiers de 0 à 4 :

```
x=0
while x<5:
    print(x)
    x+=1       /* Ne pas oublier, sinon boucle infinie ! */
```

Le grand danger est ce qu'on appelle des boucles infinies, c'est-à-dire des boucles qui ne finissent jamais. Cela arrive quand la condition est toujours satisfaite, typiquement quand on oublie d'incrémenter un compteur :

```
x=0
while x<5:
    print(x)
```

3 boucles **Tant que** pour écrire les entiers de 0 à 4.

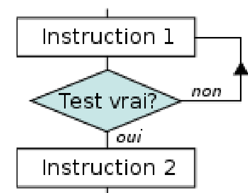
Pseudo-code	Organigramme	Scratch

x ← 0
Tant que x<5 **faire**
    écrire x
    x ← x+1
Fin tant que
	Début
	x ← 0
	x < 5 ?
	non → Fin
	oui
	Écrire x
	x ← x+1
		quand [drapeau] est cliqué
		mettre x à 0
		répéter jusqu'à ce que x = 5
		dire x pendant 1 secondes
		ajouter 1 à x

Boucle « Jusqu'à ce que »

Contrairement à un boucle « Tant que », une boucle « Jusqu'à ce que » exécute au moins une fois les instructions comprises dans la boucle.

```
REPETER
     Instruction 1
JUSQU'A CE QUE Test vrai
Instruction 2
```



Ce type de boucle n'existe pas en **Python**, mais on le trouve en **Pascal** :

```
x := 0;
repeat
    x := x+1;
    print(x)
until x=5;
```

3 boucles **Répéter jusqu'à** pour écrire les entiers de 0 à 4.

Remarquez que le bloc Scratch « répéter jusqu'à ce que » est en fait une instruction Tant que, malgré son nom.

Pseudo-code	Organigramme	Scratch
x ← 0 **Répéter** écrire x x ← x+1 **Jusqu'à ce que** x=5	Début x ← 0 Écrire x x ← x+1 x = 5 ? non / oui Fin	quand 🏁 est cliqué mettre x à 0 dire x pendant 1 secondes répéter jusqu'à ce que x = 5 ajouter 1 à x dire x pendant 1 secondes

Compteur

Un compteur permet de réaliser une boucle associée à une variable entière qui sera incrémentée (ou décrémentée) à chaque itération.

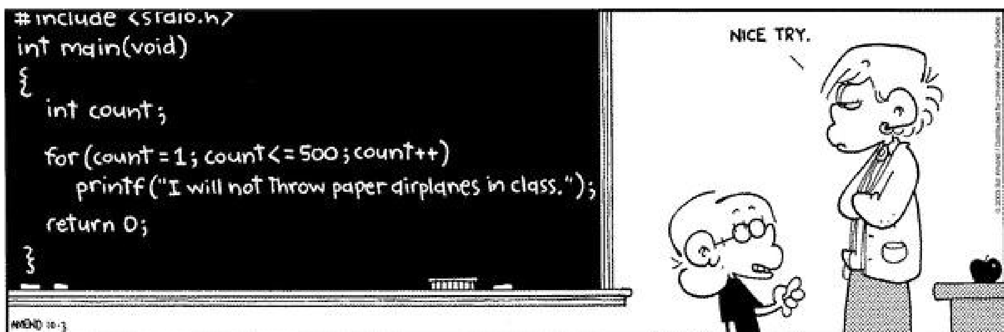
```
POUR compteur DE debut A fin
   Instruction 1
FIN POUR
Instruction 2
```

Attention ! Ici l'incrémentation est implicite ! Il ne faut pas mettre une instruction du genre compteur ← compteur + 1. Cela se fait automatiquement.

En **Pascal** :

```
for i := depart to fin do
  begin
    WriteLn(i)
  end;
```

En **C** :



Dessin de Bill **Amend**, créateur du comic strip *FoxTrot*.

En **Python** :

```
for i in range(4):
    print("i a pour valeur", i)
```

3 boucles **Pour** qui écrivent les entiers de 0 à 4.

Pseudo-code	Organigramme	Scratch
Pour x de 0 à 4 faire écrire x **Fin pour**	Début → x ← 0 → Écrire x → x ← x+1 → x = 5 ? (non → Écrire x ; oui → Fin)	quand [drapeau] est cliqué mettre x à 0 répéter 5 fois dire x pendant 1 secondes ajouter 1 à x

Itérateur

Un itérateur est un objet qui permet de réaliser une boucle parcourant tous les éléments contenus dans une structure de données (par exemple une liste).

```
POUR CHAQUE valeur DANS collection
   Instruction 1
FIN POUR CHAQUE
```

Exemple en **Python** :

```
for animal in ["chat", "chien", "poule"] :
    print(animal)
```

Exercice 6.7

Représentez sous forme d'organigramme un algorithme permettant d'afficher 500 fois « Je ne collerai plus de chewing-gum sous ma chaise. ».

Boucles imbriquées

Il est tout à fait possible, et parfois nécessaire, d'imbriquer une boucle dans une autre boucle.

Vous connaissez très bien un exemple de boucles imbriqueés : les horloges.

Par exemple, imaginons que l'on veuille écrire toutes les heures et minutes d'une journée. Cela commencera par 0 heure 0 minute, 0 heure 1 minute, 0 heure 2 minutes, ..., 0 heure 59 minutes, 1 heure 0 minute, 1 heure 1 minute, ... On voit qu'une première boucle parcourra les heures, tandis que la deuxième parcourra les minutes, et que l'on incrémentera l'heure seulement quand 60 minutes seront passées.

Voilà ce que cela donnera en Python :

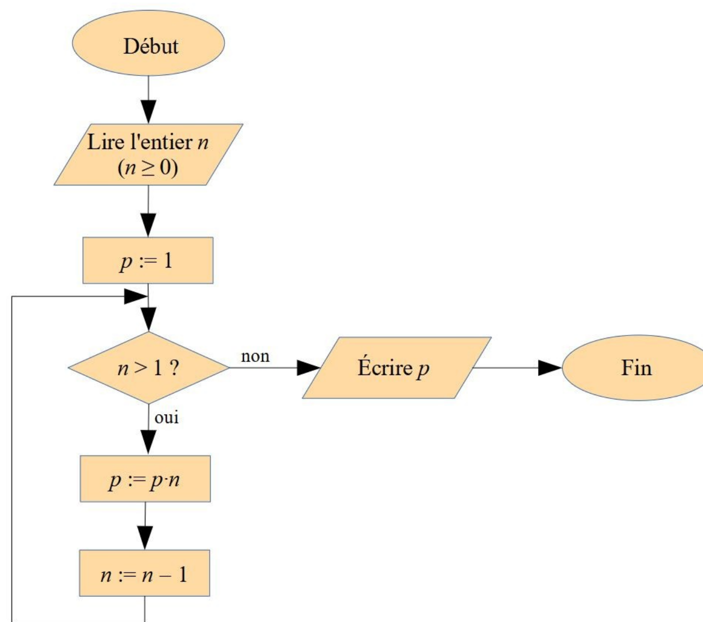
```
h=0
while h<24:
    m=0
    while m<60:
        print(h,"heure(s) ",m,"minute(s)")
        m+=1
    h+=1
```

Si l'on veut ajouter les secondes, on imbriquera une troisième boucle :

```
h=0
while h<24:
    m=0
    while m<60:
        s=0
        while s<60:
            print(h,"heure(s) ",m,"minute(s)",s,"seconde(s)")
            s+=1
        m+=1
    h+=1
```

Exercice 6.8

Déterminez ce que calcule l'algorithme ci-dessous, puis traduisez-le en pseudo-code :



Exercice 6.9

Écrivez sous forme d'organigramme un algorithme qui affiche tous les diviseurs d'un nombre naturel n donné en entrée.

Essayez de trouver l'algorithme le plus efficace possible.

Programmez-le dans le langage de votre choix (Scratch, Python, …)

Exercice 6.10

Qu'afficheront ces pseudo-codes ?

a.	b.	c.

n ← 1
tant que n < 100 **faire**
    n ← n·2
    afficher n
fin tant que
	n ← 1
	tant que n < 100 **faire**
	    n ← n^2 + 1
	    afficher n
	fin tant que
		n ← 1
		tant que n < 20 **faire**
		    n ← (n+7)%20
		    afficher n
		fin tant que

Exercice 6.11

Qu'affichera ce pseudo-code ?

C'est la suite de Syracuse...

```
n ← 20
tant que n > 1 faire:
     si n modulo 2 = 0 alors
             n ← n / 2
     sinon
             n ← 3·n + 1
     fin si
     afficher n
fin tant que
```

6.7.5. Les tableaux (listes)

Les tableaux (aussi appelés listes) sont des sortes de super variables, qui peuvent contenir plusieurs éléments (des nombres des mots. On peut voir ça comme une commode, avec des tiroirs numérotés. Chaque tiroir contiendra (ou pas) un élément.

1	2	3	4	5	6	←	indices (i)
7	3	1	11	2	8	←	liste « nombre »

Dans l'exemple ci-dessus, nombre[1] = 7.
Inversement, la valeur 2 se trouve dans la case 5.

3 algorithmes pour savoir à partir de quel indice la somme des nombres d'une liste dépasse 10.

Pour simplifier, on supposera qu'on est sûr que cela arrivera avant la fin de la liste…

Comment modifier ces algorithmes si on n'en est pas sûr ?

Pseudo-code	Organigramme	Scratch

`i ← 1`
`s ← nombre[1]`
`Tant que s ≤ 10 faire`
`    i ← i+1`
`    s ← s + nombre[i]`
`Fin tant que`
`écrire i`
	Début
	i ← 1
	s ← nombre[1]
	s > 10 ?
	oui
	Écrire i
	Fin
	non
	i ← i+1
	s ← s+nombre[i]
		quand [drapeau] est cliqué
		mettre i à 0
		mettre somme à 0
		répéter jusqu'à ce que somme > 10
		ajouter 1 à i
		ajouter élément i de nombre à somme
		dire i

Exercice 6.12

Une liste L (non triée) contient n nombres positifs. Écrivez un pseudo-code qui...

1. affiche tous les éléments de la liste.
2. affiche les éléments de la liste supérieurs à 10.
3. affiche à quel indice se trouve le nombre 42, ou dit qu'il n'y figure pas.
4. affiche le plus grand nombre de la liste, et à quel indice il se trouve. Programmez cet algorithme dans le langage de votre choix (Scratch, Python, …).
5. échange les éléments d'indices 1 et 2.

Exercice 6.13

Dessinez un organigramme qui vérifie la condition de promotion du règlement suisse sur la reconnaissance des certificats de maturité gymnasiale appelée « double compensation » :

> « Pour l'ensemble des disciplines de promotion, le double de la somme de tous les écarts vers le bas par rapport à la note 4 ne doit pas être supérieur à la somme simple de tous les écarts vers le haut par rapport à la note 4. »

La donnée est une liste *m* contenant les moyennes annuelles (entre 1 et 6, au demi-point) dans vos disciplines. Le résultat sera « condition satisfaite » ou « condition non satisfaite ».

Programmez cet algorithme dans le langage de votre choix (Scratch, Python, …).

Exercice 6.14

a. Pour diminuer la consommation d'énergie, un escalator ne se met en route que lorsqu'une personne est détectée en bas de l'escalator. Comme il faut 60 secondes pour arriver en haut, l'escalator s'arrêtera une minute après qu'une personne a été détectée.
Ecrivez un pseudo-code illustrant le fonctionnement de l'escalator.

Attention ! Il peut y avoir plusieurs personnes en même temps sur l'escalator...

b. On constate que beaucoup de personnes montent les marches de l'escalator, si bien qu'elles arrivent en haut en moins d'une minute. Faire fonctionner l'escalator durant une minute pleine n'est donc pas optimal. On décide alors d'installer un deuxième détecteur en haut de l'escalator, pour savoir si une personne est arrivée.
Écrivez un pseudo-code expliquant le nouveau fonctionnement de l'escalator.

c. En théorie, tout se passe bien. Mais en pratique ? A-t-on bien tout prévu ? N'y a-t-il pas des situations problématiques ? Si oui, comment y remédier ?
Quels sont les avantages et inconvénients de ces deux modes de fonctionnement ?

Sources

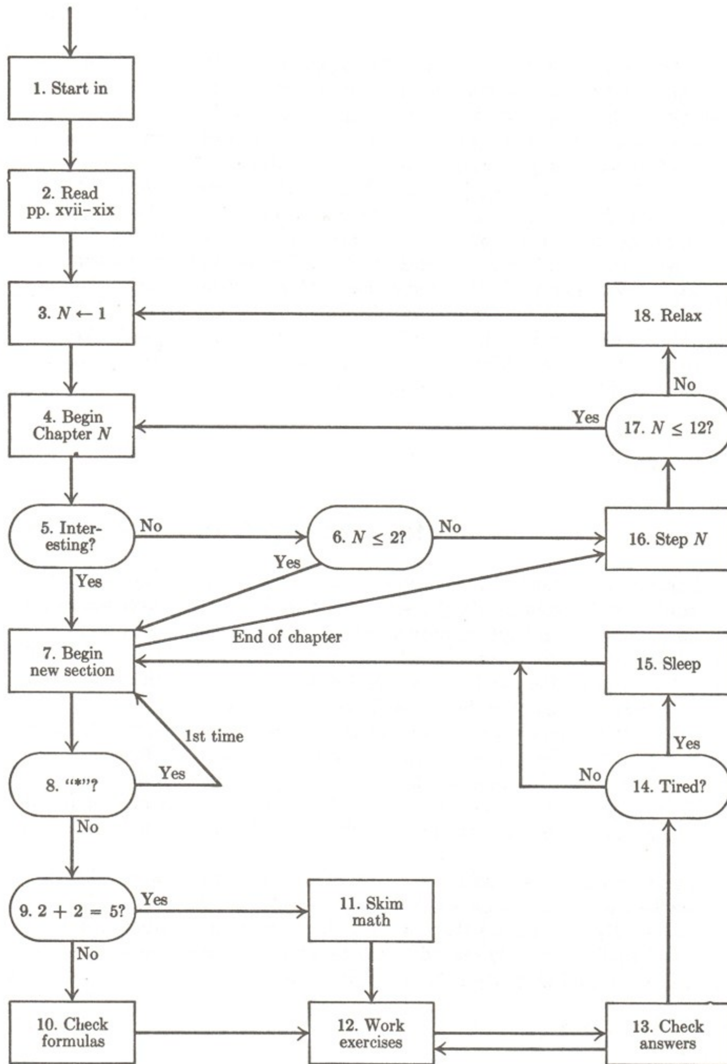
[1] Wikipédia, « Machine de Turing », <https://fr.wikipedia.org/wiki/Machine_de_Turing>

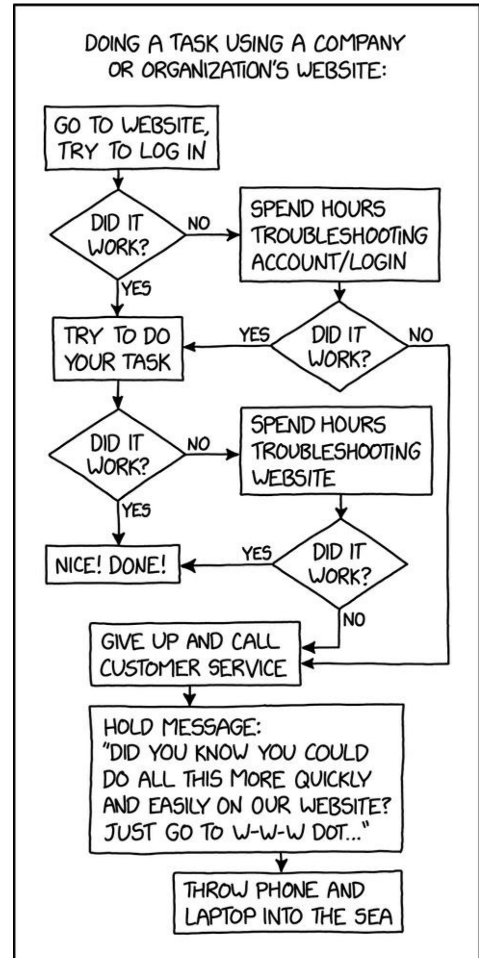
[2] Wikipédia, « Langages de programmation », <https://fr.wikipedia.org/wiki/Langage_de_programmation>

[3] Wikipédia, « Programmation informatique », <http://fr.wikipedia.org/wiki/Programmation>

[4] Wikipédia, « Paradigme (programmation) », <https://fr.wikipedia.org/wiki/Paradigme_(programmation)>

Et pour finir, un peu d'humour





The Art of Computer Programming (Vol. 1)
Donald E. Knuth, page xiv.

Website Task Flowchart
Randall Munroe, xkcd/3175