Algorithmique : Pseudo-code, table d'exécution

Chapitre 1

I- Algorithme

1) Histoire

On peut donner différentes définitions d'un algorithme en voilà une :

Définition 1 :

Un **algorithme** est une suite d'instructions élémentaires appliquées dans un ordre déterminé portant sur un nombre fini de données pour arriver, en un nombre fini d'étapes, à un certain résultat.

Quelques repères historiques :

Les premiers algorithmes ont été développés bien avant l'émergence de l'informatique et même bien avant le grand savant Muhammad ibn Musa al-Khwârizmî dont le nom latinisé a donné le mot algorithme.
De plus, vous manipulez des algorithmes depuis votre prime enfance : une recette de cuisine est un exemple concret d'algorithme !

- **Vers -1800 :**

Les plus anciens algorithmes connus remontent il y a presque quatre millénaires.
Les **Babyloniens** qui vivaient en Mésopotamie (actuel Irak) utilisaient des algorithmes pour résoudre certaines équations (comme celles du second degré).
Voici l'image d'une tablette datant de cette période où plusieurs problèmes du second degré sont résolus par une sorte de liste d'instructions proche de nos algorithmes actuels :

- **Vers -300 :**

Euclide a proposé, entre autre un algorithme, encore utilisé de nos jours, permettant le plus grand commun diviseur (le PGCD) entre deux nombres entiers. Vous avez vu cet algorithme d'Euclide au collège.

- **Vers 800 :**

Le mot algorithme vient du nom latinisé du grand mathématicien **Al-Khwârizmî**.

Ce savant ayant vécu entre 780 et 850 fut membre de la Maison de la Sagesse de Bagdad. Il répertoria les algorithmes connus à son époque et, entre autres travaux, il fut l'auteur de deuxlivres importants :

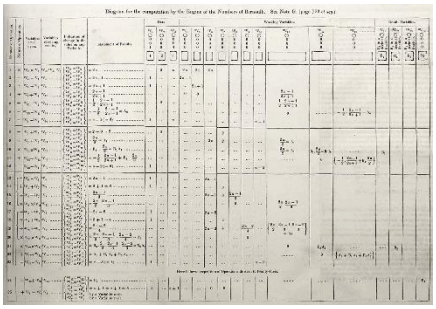
- le premier a conduit au mot « algèbre » actuel ;
- le second a permis la diffusion du système de numération décimal actuel à travers le monde abbasside puis en Europe : ce sont les « chiffres arabes » actuels.

- ## XVII siècle :

Afin de réduire le temps de calcul et surtout les risques d'erreurs de calcul, à partir du XVII siècle, des calculateurs mécaniques ont été construits. Voici l'image de la toute première calculatrice construite par Blaise Pascal en 1645 capable d'effectuer des additions et des soustractions : **La Pascaline**

- ## XIX siècle :

Exaspéré par les nombreuses erreurs présentes dans les tables utilisées pour faire des calculs compliqués en sciences (astronomie, physique, ...), l'anglais Charles Babbage conçoit les plans d'une machine capable de calculer puis d'éditer les valeurs de fonctions polynomiales.
Ada Lovelace, la fille du poète Lord Byron, travaille un temps avec Charles Baggage et écrit en 1843 le premier algorithme exécutable sur une machine : c'est le **premier programme informatique** !



- ## 1936 :

Le concept de **machine universelle**, capable d'exécuter tous les algorithmes est développé par Alan Turing. Les notions de machine, d'algorithme, de langage et d'information sont pensées désormais comme un tout cohérent.

- ## 1943 :

La **première machine électronique**, le Colossus, a été construite en 1943 en Angleterre et a été utilisé pour décrypter les codes secrets allemands fondés sur la machine Enigma.

- ## 1948 :

Le **premier ordinateur** suivant l'architecture de Von Neumann est construite aux États-Unis. C'étaient surtout des femmes qui travaillaient dans la programmation des premiers ordinateurs. Le seul langage directement utilisable par le processeur des ordinateurs est le langage machine (abordé brièvement plus tard cette année).
Pour faciliter la communication d'informations avec un ordinateur, des informaticiens ont créé des **langages** dits de haut niveau qui sont plus simples à utiliser, car plus proches du langage naturel. Il y en a un très grand nombre (FORTRAN (1955), C (1972), PHP (1994), JAVA (1995), Javascript (1995), ...)
Celui que vous utiliserez énormément cette année est le langage PYTHON, créé en 1991 par Guido Von Rossum.

2) Pseudo-code

Puisqu'il y a un très grand nombre de langages différents, il est commode d'utiliser une sorte de *lingua franca* qui permet d'écrire un algorithme dans un langage "universel". Le **pseudo-code** sert à cela !

Le pseudo-code est un langage pour exprimer clairement et formellement un algorithme.
Ce langage est près d'un langage de programmation comme Pascal, C# ou C++, sans être identique à l'un ou à l'autre. Il exprime des idées formelles dans une langue près du langage naturel de ses usagers (pour nous, le français) en lui imposant une forme rigoureuse.
Il n'y a pas de standard normalisé mais seulement des conventions partagées par un plus grand nombre de programmeurs.

Quelques règles :

- Le nom d'une variable ou d'une constante doit être significatif. On devrait savoir immédiatement, à partir de son nom, à quoi sert la variable ou la constante, et quel sens donner à sa valeur
- Les majuscules et les minuscules sont des symboles distincts dans la plupart des langages de programmation, mais pas tous. Ainsi, pour éviter les ennuis, ne donnez pas à deux entités (une variable et une constante, par exemple) des noms qui ne différeraient que sur cet aspect
- Les instructions se font une ligne à la fois (pas de ' ; ' en pseudo-code)
- On ne se préoccupe pas des types des variables et des constantes (ce principe n'est pas universel)
- Les opérations de base sont LIRE, ÉCRIRE et ← (affectation d'une valeur à une variable). Les opérations de base et/ou mots clés doivent être écrits en gras ou en majuscules

Exercice 1 :

Corriger le pseudo-code suivant :

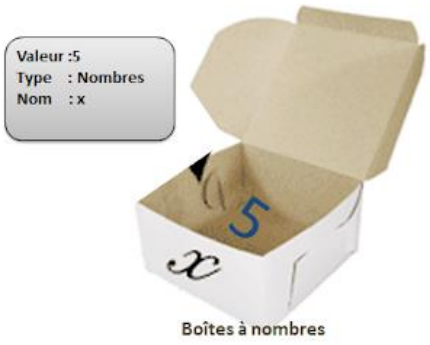
```
1. a et B sont des entiers

2. A ← 4 ; LIRE B

3. C'est une variable ← A+B;

4. ÉCRIRE c'est une variable
```

II- Les Variables

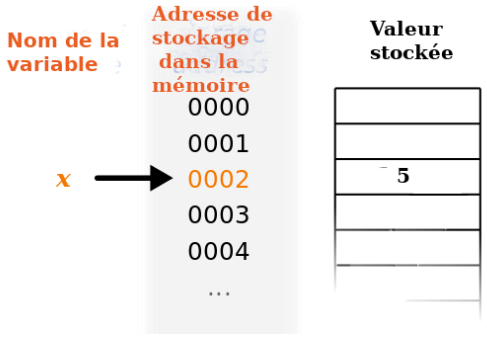
1) A quoi servent les variables ?

Dans un programme informatique, on va avoir en permanence besoin de stocker provisoirement des valeurs. Il peut s'agir de données issues du disque dur, fournies par l'utilisateur (frappées au clavier), ou que sais-je encore. Il peut aussi s'agir de résultats obtenus par le programme, intermédiaires ou définitifs. Ces données peuvent être de plusieurs **types** (on en reparlera) : elles peuvent être des nombres, du texte, etc. Toujours est-il que dès que l'on a besoin de stocker une information au cours d'un programme, on utilise une **variable**.

Pour employer une image, une variable est une boîte, que le programme (l'ordinateur) va repérer par une étiquette. Pour avoir accès au contenu de la boîte, il suffit de la désigner par son étiquette.
Ainsi, ci-dessous la variable xx est l'étiquette d'une boîte contenant l'entier 5.



En réalité, dans la mémoire vive de l'ordinateur, il n'y a bien sûr pas une vraie boîte, et pas davantage de vraie étiquette collée dessus (j'avais bien prévenu que la boîte et l'étiquette, c'était une image). Dans l'ordinateur, physiquement, il y a un emplacement de mémoire, repéré par une adresse binaire. Si on programmait dans un langage directement compréhensible par la machine, on devrait se fader de désigner nos données par de superbes 0010 (souvent écrit en hexadécimal 0002) ou 10011011 (enchanté !).
Ainsi ci-dessous, la variable nommée x fait référence à l'emplacement de mémoire dont l'adresse en hexadécimale est 0002. Cet emplacement mémoire stocke le nombre 5.



Mauvaise nouvelle : de tels langages existent ! Ils portent le doux nom d'assembleur.
Bonne nouvelle : ce ne sont pas les seuls langages disponibles.

Les langages informatiques plus évolués (ce sont ceux que presque tout le monde emploie) se chargent précisément, entre autres rôles, d'épargner au programmeur la gestion fastidieuse des emplacements mémoire et de leurs adresses. Et, comme vous commencez à le comprendre, il est beaucoup plus facile d'employer les étiquettes de son choix, que de devoir manier des adresses binaires.

2) Déclaration des variables et affectations

Il existe de nombreux types de variables utilisés :

types	Remarques
booléen	vrai ou faux
caractère	symbole typographique
chaine de caractères	ensemble de caractères entre " "
entier	entiers relatifs
flottant	"Utilisé pour les réels"

Il existe d'autres types de variables. Le fait de déclarer en pseudo-code le type de la variable n'est pas une obligation, mais vous le verrez dans de nombreux sites car de nombreux langages de programmation nécessitent un typage strict.

Le langage PYTHON est auto-typé, c'est-à-dire le typage se fait lors de l'affectation.

Comme vu précédemment, en pseudo-code, l'instruction d'affectation se note avec le signe ←

```
1. toto ← 40
```

On attribue la valeur 40 à la variable toto

III- Les instructions

1) Test IF THEN ELSE

Cette instruction conditionnelle teste une expression booléenne (condition après le if) et exécute soit un bloc d'instruction(s) si la condition est vraie (bloc 1 après le then) soit un éventuel autre bloc d'instruction(s) si la condition est fausse (bloc 2 après le else).

```
1 if condition then
2       bloc 1
3 else
4       bloc 2
```

Voici une manière imagée de visualiser cette instruction :

Remarques :

- La partie else n'est pas obligatoire. On peut utiliser l'instruction if ...then,
- Il existe de nombreuses variantes pour indiquer les blocs d'instructions :
    - L'indentation qui se caractérise par le bloc décalé.

      Ce principe sera utilisé en langage PYTHON.

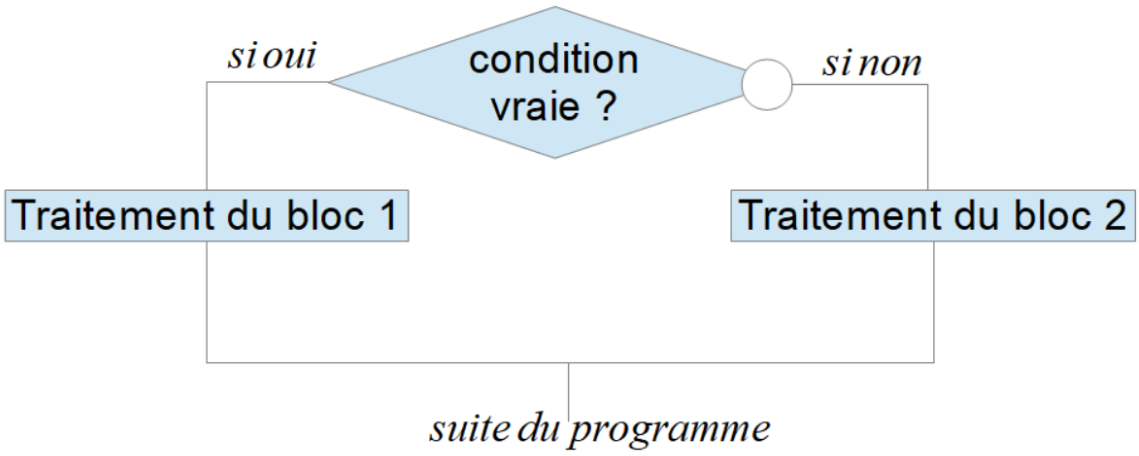
    - Une fin de bloc du type FinDeSi,
    - Des mots clés du type début et fin.

Vous pouvez pratiquer du pseudo-code en français ou utiliser l'anglais (il faut vous habituer à lire du pseudo-code en anglais).

Une version francisée avec une identification des blocs différents pourrait être :

```
si condition alors
        bloc 1
sinon
        bloc 2
```

Une autre manière de visualiser cette instruction, sous forme d'algorigramme :



Exemple 1 :

Voici un algorithme affichant le signe d'un nombre a saisi par l'utilisateur :

```
LIRE a
if a>0 then
        ÉCRIRE "a positif"
else
        ÉCRIRE "a négatif"
```

2) Boucle FOR

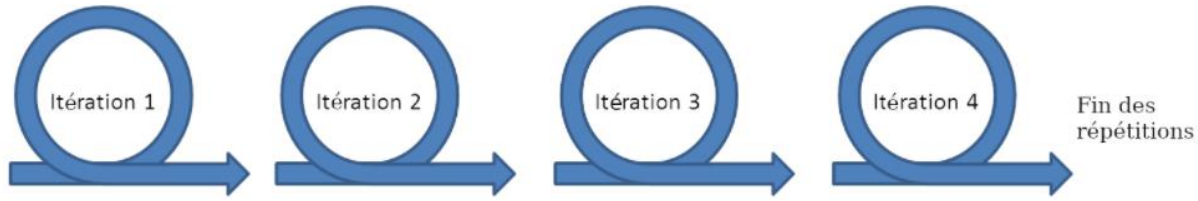
L'instruction **Pour** est utilisée lorsque le nombre d'itérations est connu à l'avance : elle initialise un compteur, l'**incrémente** (c'est-à-dire l'augmente de 1) après chaque exécution du bloc d'instructions, et vérifie que le compteur ne dépasse pas la borne supérieure.

```
for compteur←entier1  to  entier2
        bloc d'instructions
```

On peut préciser le pas d'incrémentation du compteur. Par défaut le pas est de 1. Le mot clé en anglais pour le pas est le mot **STEP**.

Voici une illustration de l'algorithme suivant :

```
1 for i←1  to  4
2       bloc d'instructions
```



Exemple 2 :

Voici un algorithme affichant les 10 premiers nombres entiers positifs ; ceux de 0 à 9 :

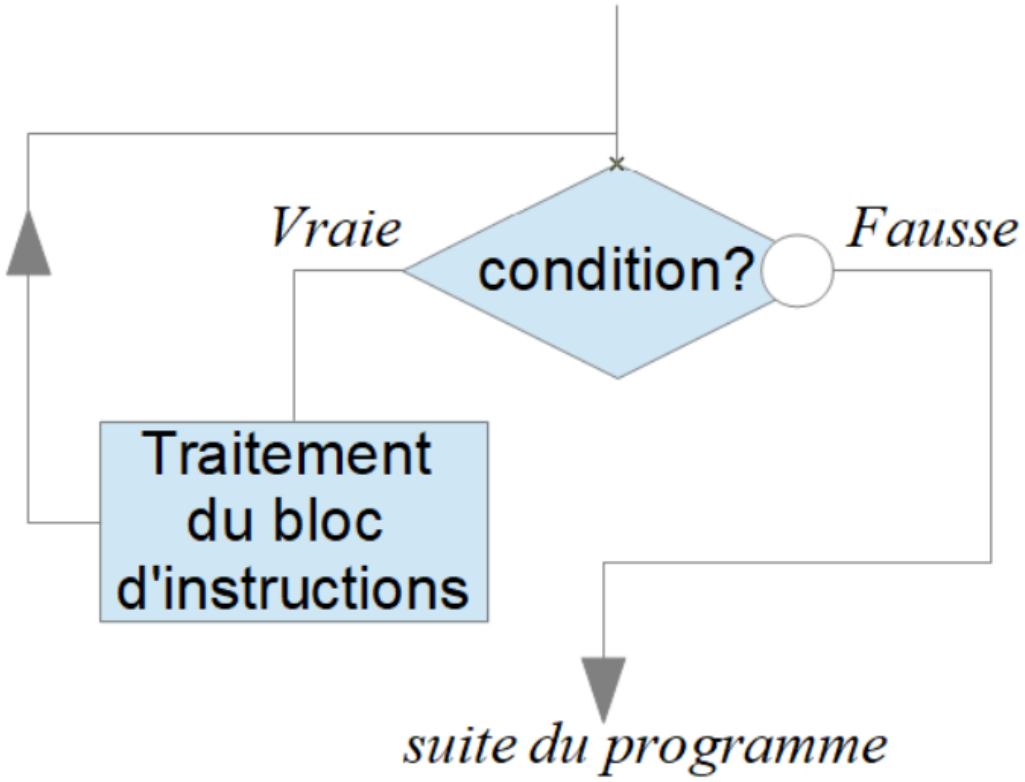
```
1 for i←0  to  9
2       ÉCRIRE i
```

3) Boucle WHILE

L'instruction **While** est utilisée lorsque le nombre d'itérations n'est connu à l'avance : elle répète l'exécution le bloc d'instructions tant que la condition est vraie.

```
1 while  condition vraie
2       bloc d'instructions
```

Une manière de visualiser cette instruction, sous forme d'algorigramme :



Exemple 3 :

Voici un algorithme affichant tous les carrés inférieurs à 50 :

```
1 i ← 0
2 while  i*i < 50
3       ÉCRIRE i*i
4       i ← i+1
```

Remarques :

- Faire attention à éviter une boucle infinie !
- Dans l'exemple précédent l'incrémentation de i assure la fin de la boucle WHILE.
- De plus, si la condition n'est pas vraie initialement, aucune itération de la boucle WHILE n'a lieu.

IV- L'importance des commentaires

Il faut prendre l'habitude de commenter ses algorithmes et ses programmes.

Remarques :

- cela permet une relecture facile du code
- cela permet une lecture plus facile pour une personne tierce (un correcteur par exemple)
- commenter n'est pas paraphraser. Le commentaire doit apporter une information

Exemple 4 :

Voici l'algorithme sans les commentaires **:**

```
1 n←0
2 p←50
3 while  p<100
4       p←p*1.1
5       n←n+1
6 ÉCRIRE n
```

Il existe différentes façons de noter un commentaire : //, #, <- ->

Voici l'algorithme avec les commentaires :

```
// Cet algorithme cherche la valeur de n pour que p passe de 50 à 100 avec une augmentation de 10%
1 n←0                          //Initialisation de n
2 p←50                         //Initialisation de p
3 while  p<100
4       p←p*1.1                //p suivie de 10 % d'augmentation
5       n←n+1                  //Incrémentation de n
6 ÉCRIRE n
```

Avec trop de commentaires, le code devient illisible. Sans commentaire, le code est incompréhensible. A vous de trouver le juste milieu.

Exercice 2 :

Commenter le code suivant :

```
1 S←0
2 for  i←1 to 10
3       S←S+1
4 ÉCRIRE S
```

V- Table d'exécution d'un algorithme

Il existe différentes manières de réaliser une trace de programme et/ou d'algorithme. Une trace :

- permet de **suivre pas à pas** l'algorithme;
- permet de **détecter des erreurs**;
- permet de **contrôler** que l'algorithme fait bien ce que l'on avait prévu;
- permet de **comprendre** ce que fait un algorithme.

Dans la mesure du possible, on peut organiser une **trace d'exécution** d'un algorithme en constituant un tableau avec toutes les variables de l'algorithme. Il faut numéroter toutes les lignes de votre algorithme. En colonne, il faut indiquer le nom des variables et en ligne les numéros de ligne.

Exemple 5 :

```
r←0
while r*r<= n
       r←r+1
r←r-1
```

Il faut commencer par numéroter toutes les lignes de l'algorithme.

```
1 r←0
2 while r*r<= n
3       r←r+1
4 r←r-1
```

Voici une trace de l'algorithme avec n=5. Quelle est la valeur de la variable r ?

#ligne	n	r	Commentaires
1	5	0	Initialisation
2	5	0	0*0 <=5 , on entre dans la ligne 3
3	5	1	r←1
2	5	1	1*1 <=5 , on entre dans la ligne 3
3	5	2	r←2
2	5	2	2*2 <=5 , on entre dans la ligne 3
3	5	3	r←3
2	5	3	3*3>5 , on sort de la boucle
4	5	2	r←2

À la fin de l'algorithme, la variable r a pour valeur 2

En mathématiques, vous auriez une version minimaliste de ce tableau, qui correspondrait à l'état des variables :

n	5	5	5	5	5	5
r	0	0	1	2	3	2

Nous allons favoriser des traces de ce type :

```
#Algorithme Algo1
#Déclarations
#Variables
#a, b, c, resultat : entier
#Début
#{1}
a = -4
#{2}
b = 3
#{3}
c = a + b
#{4}
a = b - c
#{5}
c = b
```

```
#{6}
if a > c :
#{6a.1}
   b = a + b
else :
#{6b.1}
   c = b - 2*a

#{7}
resultat = 3*b - 2*c+2
```

Inst.	*Contrôle*	*Variables locales*			
		a	*b*	*c*	*resultat*
Avant		*?*	*?*	*?*	*?*
1	×	*-4*			
2	×		*3*		
3	×			*-1*	
4	×	*4*			
5	×			*3*	
6	*(a>c) ? Vrai*				
6a.1	×		*7*		
7	×				*17*

Exercice 3 :

En vous inspirant de l'exemple ci-dessus, réaliser une trace de l'algorithme précédent avec n=20.

Pour vous aider voici un algorithme (donné en évaluation) avec sa trace :

```
1 i←0
2 j←n-1
3 p←Vrai
4 while i<=j
5       if mot[i]=mot[j] then
6               i←i+1
7               j←j-1
8       else
9               p←Faux
10              i←1
11              j←0
12 Afficher p
```

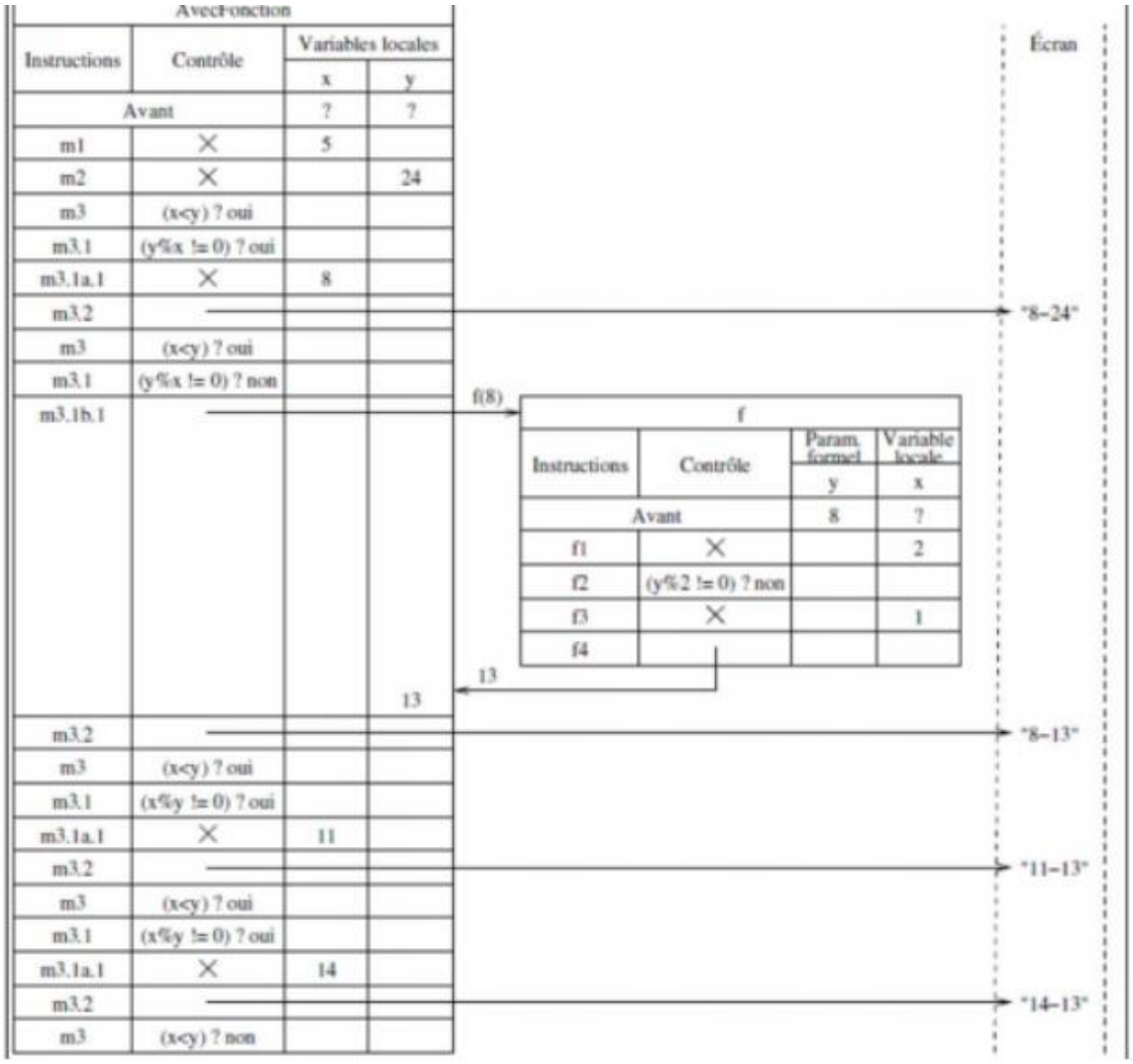
1. Effectuer la trace d'algorithme avec `mot=[r,a,d,a,r]`
2. Effectuer la trace d'algorithme avec `mot=[n,s,i]`

Instructions	Contrôle	Variables locales				
		i	J	n	P	mot
Avant		?	?	5	?	[r,a,d,a,r]
1		0				
2			4			
3					V	
4	I<=j V					
5	Mot[i]=mot[j] V					
6		1				
7			3			
4	I<=j V					
5	Mot[i]=mot[j] V					
6		2				
7			2			
4	I<=j V					
5	Mot[i]=mot[j] V					
6		3	1			
4	I<=j F					
12					V	

Un autre exemple avec un appel de fonction :

```
def f (y : int) -> int :
    #Déclarations
    #Variables locales
    #x : entier
    #Début
    #{f1}
    x = y / 4
    #{f2}
    while y % 2 != 0 :
        #{f2.1}
        y = y + 7

    #{f3}
    x = 3 * x - y + 3
    #{f4}
    return (y + 5 * x)
    #Fin
```

```
#Algorithme
#Déclarations
#Variables
#x, y : entier
#Début
#{m1}
x = 5
#{m2}
y = x * x - 1
#{m3}
while x < y :
#{m3.1}
    if y % x != 0 :
        #{m3.1a.1}
        x = x + 3
    else :
        #{m3.1b.1}
            y = f(x)
#{m3.2}
    print(x , "-" , y)
#Fin
```



VI- Exercices

Exercice 4 :

Écrire un algorithme en pseudo code qui calcule la moyenne de trois nombres a, b et c. Le résultat sera stocké dans une variable m.

Exercice 5 :

Écrire un algorithme qui renvoie le max de deux nombres a et b. Le résultat sera stocké dans une variable max.

Exercice 6 :

Écrire un algorithme qui demande un nombre entier non nul de départ, et qui calcule la somme des entiers jusqu'à ce nombre.
Par exemple, si l'on entre 5, le programme doit calculer : 1 + 2 + 3 + 4 + 5 = 15.

Exercice 7 :

Écrire un algorithme qui stocke dans une variable max le maximum de trois variables a, b et c données.

Exercice 8 :

Trouver l'erreur dans l'algorithme suivant :

```
1 for truc ← 1 to 10
2       truc ← truc * 2
```

Exercice 9 :

Trouver l'erreur dans l'algorithme suivant :

```
1 while i > 0
2       i ← i - 1
```

Exercice 10 :

Il existe un type list en python (en fait c'est plutôt un type tableau en informatique).
Pour définir un tableau/liste on écrira les éléments entre crochets :

```
liste1 ←[3,5,2,14,8]
```

Le premier élément de la liste est le nombre 3 et le dernier élément est le nombre 8. Il y a 5 éléments dans cette liste.
Attention les cases sont numérotées à partir de 0. L'indice du nombre 3 est le nombre 0. L'indice du nombre 8 est le nombre 4. Nous verrons plus tard quelle en est la raison.
On peut utiliser les notations suivantes : liste1[0]=3 ; liste1[1]=5 ; ……. ; liste1[4]=8

1. Écrire un algorithme en pseudo code qui renvoie le maximum max des éléments d'une liste nommée liste1 ayant n éléments, liste saisie par un utilisateur.
2. Écrire une trace d'exécution en prenant la liste1 de l'exemple.

Exercice 11 :

Écrire un algorithme qui permet d'échanger le contenu de deux variables a et b.

Exercice 12 :

Après avoir réalisé la trace de cet algorithme avec a=17 et b=3, précisez ce que représentent a et i après exécution de algorithme.

```
1 i←0
2 u←b
3 while a>=b
4       a←a-u
5       i←i+1
```

Exercice 13 :

Écrire un algorithme qui dit si un nombre appartient à une liste a n éléments. En sortie l'algorithme renvoie un booléen : True signifiant que la valeur est dans la liste, False sinon.

Exercice 14 :

Voici un algorithme :

```
1 nb ← 0
2 n ← longueur de la liste
3 for i ← 0 to n-1
4       if liste[i]='a' then
5               nb ← nb + 1
```

1)

a. Exécuter l'algorithme avec la liste liste=['r', 'a', 'd', 'a', 'r']

b. Expliquer ce que fait cet algorithme.

c. Exécuter l'algorithme avec la liste liste=['e', 's' , 't'].

2) Transformer l'algorithme pour qu'il teste la présence de la lettre 'a'.

3) On peut utiliser une instruction in . Cette instruction est utilisée en PYTHON. Par exemple :

for elt in ['a', 'e', 'i']

va boucler pour tous les éléments de la liste. elt va prendre la valeur 'a', puis ensuite la valeur 'e', puis ensuite la valeur 'i'.
En utilisant cette nouvelle instruction, transformer l'algorithme pour qu'il teste la présence d'une voyelle.

4) Transformer l'algorithme pour qu'il compte le nombre de voyelles.

Exercice 15 :

En pseudo-code, comme en Python, une chaîne de caractères est représentée par un ensemble de glyphes encadrés par " ou par ''.
Par exemple : "Il fait beau !" ou 'A5x@f.r' sont deux chaînes de caractères.

On peut accéder directement à un caractère de la chaîne en utilisant sa position de la chaîne en partant de 0.
Par exemple : si ch="Il fait beau !" alors ch[0]='I' , ch[4]='a' (car l'espace compte comme un caractère aussi), etc

On notera ici, comme dans le langage Pyhton, la concaténation de deux caractères 'a' et 'b' ainsi : 'a'+'b', ce qui donne comme résultat 'ab'

1. Proposer une table d'exécution pour l'algorithme suivant : (rq : <> signifie « différent de «)

```
1 phrase ="arret"
2 lettre ← 'r'
3 fin ← ''
4 for elt in phrase
5       if elt <> lettre then
6               fin ← fin + elt
```

2. L'algorithme précédent a été modifié. Que fait ce nouvel algorithme présenté ci-dessous ?

```
1 phrase ="C'est un trou de verdure où chante une rivière."
2 lettre ← 'r'
3 fin ← ''
4 for elt in phrase
5       if elt <> lettre then
6               fin ← fin + elt
```

3. Comment modifier l'algorithme précédent pour que la variable fin soit égale à "rrrrr"

Exercice 16 :

1. Faire une trace de l'algorithme suivant avec liste = [3,2,5,4,1] :

```
1 PG ← 0
2 IPG ← 0
3 n ← longueur de la liste
4 for i← 0 to n-1
5       if liste[i] > PG then
6               PG ← liste[i]
7               IPG ← i
```

2. Que fait cet algorithme ? Que représentent les variables PG et IPG ?

VII- Exercices de renforcements

Exercice 17 :

L'algorithme ci-dessous a pour but de calculer le prix à payer en fonction du nombre d'objets acheté d'un produit. Le produit coûte 2€50 pièce. Corriger le pseudo-code suivant :

```
1 A est un entier et B est un réel.
2 Lire A
3 B prend la valeur 2.50A
4 ÉCRIRE B
```

Exercice 18 :

Écrire un algorithme en pseudo code qui calcule la moyenne de cinq nombres a, b, c, d et e. Le résultat sera stocké dans une variable moy.

Exercice 19 :

Écrire un algorithme qui stocke dans une variable min le minimum de deux variables a et b données.

Exercice 20 :

La factorielle d'un nombre entier non nul (factorielle se note avec un !).
Par exemple 4! = 4×3×2×1 = 24
Ainsi, 1! = 1 et pour n > 1 ; n! = n×(n−1)×...×2×1 = n×(n−1)!
Écrire un algorithme qui demande un nombre entier n non nul de départ, et qui calcule n!, c'est-à-dire le produit des entiers jusqu'à ce nombre n.

Exercice 21 :

Écrire un algorithme qui stocke dans une variable min le minimum de trois variables a, b et c données.

Exercice 22 :

Pour définir un tableau/liste on écrira les éléments entre crochets :

```
liste1 ←[7,4,3,8,9]
```

Le premier élément de la liste est le nombre 7 et le dernier élément est le nombre 9. Il y a 5 éléments dans cette liste.
Attention les cases sont numérotées à partir de 0. L'indice du nombre 3 est le nombre 0. L'indice du nombre 9 est le nombre 4. Nous verrons plus tard quelle en est la raison.
On peut utiliser les notations suivantes :
liste1[0]=7 ; liste1[1]=4 ; ……. ; liste1[4]=9

1. Écrire un algorithme en pseudo code qui renvoie le minimum min des éléments d'une liste nommée liste1 ayant n éléments.
2. Écrire une trace d'exécution en prenant la liste1 de l'exemple.

: