CORRIGÉ

[Stéphane BEAUDET – Frédéric PEURIERE]

PREMIERS PAS EN ALGORITHMIQUE

1er pas d'ALGORITHMIQUE

A - DÉFINIR : Un **algorithme** est une suite finie et non ambiguë d'opérations ou d'instructions permettant de résoudre une classe de problèmes (du nom du mathématicien perse Al-Khwârizmî vers 780)

Ainsi, cette notion est très adaptée à l'informatique (l'ordinateur étant le réceptacle de ces instructions)

B- RÉDACTION : Il peut être écrit en langage « naturel » ou en pseudo code et **implémenté (**=traduit) dans différents langages : Python,Javascript, calculatrice...

Quelques exemples d'instructions :

Langage « naturel »	Pseudo code	Python	Javascript	TI // CASIO	
Affecter à A la valeur 5	A ← 5	A=5	var A=5	5→A (Pour TI : Touche STO>)	
Afficher A	Afficher A	print(A)	alert(A) / console.log(A)	Disp A	A◢
Saisir A	Saisir A	A=float(input ('A='))	Prompt A	Prompt A	"A=" ?→A↵

Si *Condition* **Alors faire**
Instructions1
Sinon faire
Instructions2
	Si *Condition* **Alors**
	Instructions1
	Sinon
	Instructions2
	FinSi
		if *Condition* **:**
		Instruction1
		else :
		Instruction2
			if (condition1) {
			instructions 1; }
			else {
			instructions2;}
				:**If** *condition*
				:**Then**
				:*Instruction1*
				:Else
				:*Instruction2*
				:End
					If *condition*↵
					Then *Instruction1*↵
					Else *Instruction2*↵
					IfEnd↵

Pour i allant de 3 à 7 par pas de 1 **faire** *Instructions*	**Pour** i allant de 3 à 7 *Instructions* **FinPour**	***for*** *i in range (3,8) :* *Instructions*	**for** (var i=3 ; i<8 ; i++) { instructions ; }	**:For**(*i*,3,7) **:***Instructions* **:End**	**For** 3→*i* **To** 7↵ *Instructions*↵ **Next**
Tant que la condition est vraie **faire** *Instructions*	**Tant que** Condition est vraie **faire** *Instructions* ***Fin du Tant que***	**while** *condition :* *Instructions*	**while** (condition) { Instructions} ;	**:While** *Condition* **:***Instructions* **:End**	**While** *Condition*↵ *Instructions*↵ **WhileEnd**↵

Citez une instruction qui ne s'écrit jamais exactement de la même manière dans aucun des 5 langages : L'affectation de valeur a une variable

Citez quelques différences et points communs de ces langages : une conditionnelle, Si, s'ecrit toujours « if » dans les langages de programmation mais le Alors, lui peut etre traduit par « : »(python) ou « » (javascript) ou « then » (calculatrices)

De même pour les boucles Tant que et Pour

C- COMPLEXITÉ TEMPORELLE

En informatique, la question de performance est centrale (surtout si ce temps utile pour obtenir un résultat est « long »). De manière générale, le traitement de données dépend du volume de ces données et de la nature du traitement que l'on fait.

Par exemple, une boucle peut posséder un grand nombre de répétitions →

```
n=100
for i in range (n):
    print (i)
```

le programme affiche 100 lignes

mais l'effet est encore démultiplié avec des boucles imbriquées. →

```
n=100
for i in range (n):
    for j in range (n):
        print(i,j)
```

le programme affiche 100^2 lignes

Combien de lignes afficherait ce nouveau programme ?

```
n=100
for i in range (n):
    for j in range (n):
        for k in range(n):
            print(i,j,k)
```

$100x100x100=100^3= 10^6$ (1 million)

On multiplie maintenant n par 10, donner le coefficient de multiplication du nombre de lignes dans chacun de ces 3 programmes.

1er programme: le nombre de lignes est multiplie par 10 , par 10 x10 = 100 au 2eme programme, et par 10x10x10 = 1000 au 3eme

On parle alors de **complexité TEMPORELLE** de l'algorithme : on s'attache rarement à la détermination exacte de ce temps (on conserve une estimation)

*Remarque : bien que ce cours se limite à cette complexité temporelle, il existe aussi une **complexité spatiale** qui rend compte de l'espace mémoire pendant l'exécution*

Pour mesurer la complexité temporelle, nous avons déjà vu quelques instructions Python de mesure du temps.

```
from time import time
debut = time()
# Code dont on mesure le temps
fin = time()
print("Temps passé : ",fin - debut)
```

Évaluez la durée des 3 programmes ci dessus.

Les réponses sont différentes selon les performances du calculateur interne (CPU):

EXEMPLE dans mon ordi perso	P1	P2	P3
Temps d'exécution en secondes	0.1882	15.8556	3006.5999 = 50 minutes !

x100 x190

D - VALIDITÉ : CORRECTION et TERMINAISON d'un algorithme

Être certain qu'un algorithme donne le bon résultat est essentiel !!

1°) La **terminaison** stipule que les calculs décrits par l'algorithme s'arrêteront. Évidemment, savoir déceler cela avant toute implémentation évite le « plantage » de la machine.

On utilise souvent un **variant de boucle** : c'est une fonction entière, positive, qui décroit strictement

Il vérifie que la boucle se termine: f(i)=0 => !B (non B, ou B est un booléen)

2°) Afin de déterminer la **correction** d'un algorithme, on détermine un **invariant de boucle** : c'est une propriété ou une formule logique qui doit être vraie, à l'initialisation et à chaque itération de la boucle quel que soit le nombre d'itération. La difficulté réside donc à le trouver.

Exemple : dans un tableau de n entiers, trouver le plus grand élément.

```
def maximum(liste):              # recherche le maximum dans la liste
    n = len(liste)               # longueur de la liste
    i=0                          # le premier indice
    maxi = liste[0]              # on démarre au premier élément de la liste
    while i < n:                 # on parcourt la liste (i indice d'un élément de la liste)
        if liste[i]>maxi:        # la valeur rencontrée est supérieur à maxi
            maxi = liste[i]      # on affecte une nouvelle valeur à maxi
        i=i+1                    # on s'occupe de l'indice suivant
    return maxi                  # la liste est parcourue : maxi contient le maximum
```

variant de boucle : f(i) = n-i

invariant de boucle : liste non modifiée ET maxi est le plus grand de liste[0....i] ET $0 \leq i \leq n$

Compléter l'algorithme suivant

```
q=0
n=      # a compléter
while n!=0:
    n-=3
    q+=1
print(q)
```

… pour qu'il termine. n= 18

… pour qu'il ne termine pas. n=10 ou n=-18

Déterminer la condition de terminaison sur le variant de boucle n : l'algorithme se termine si n est un multiple positif de 3

Donner un invariant de boucle pour la fonction suivante qui calcule x à la puissance n :

```
def puissance(x,n):
    r=1
    for i in range(n):
        r=r*x
    return r
```

Avant d'exécuter le tour de boucle i, on a $r=x^i$. On vérifie que c'est bien vrai pour i=0 car $r=1=x^0$

On peut donc écrire **for i in range(n):**

 # *invariant :* r=x à la puissance i

E- ALGORITHMES DE RECHERCHE

ALGORITHMES DE TRI

Trier un tableau de nombres, c'est ranger ces nombres dans l'ordre croissant. Par exemple, le tableau [6 3 7 2 3 5] devient une fois trié [2 3 3 5 6 7] . Bien sûr, on a souvent besoin de trier d'autre choses que des nombres (comme des mots par ordre alphabétique, des fichiers par longueur, des messages par date, les résultats d'une recherche par pertinence, des articles par référence, des personnes par date de naissance...), mais les algorithmes utilisés sont les mêmes. Contrairement à ce qui se passe en général dans la vie courante, en informatique, on trie de très grandes quantités de données (quelques centaines de milliers d'éléments) à tout bout de champ. C'est pourquoi les informaticiens ont inventé de très nombreuses méthodes de tri, plus ou moins rapides et efficaces. Les ordinateurs passent énormément de temps à faire des tris : on considère aujourd'hui que les algorithmes de tri sont ceux qui sont les plus utilisés par les ordinateurs du monde entier !