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 : ….....

Citez quelques différences et points communs de ces langages : .....

.....

.....

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)
```

….....

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

….....

.....

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...Comparez avec vos camarades

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: $f(i)=0 \Rightarrow !B$ (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.

… pour qu'il ne termine pas.

Déterminer la condition de terminaison sur le variant de boucle n : ….....

….....

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
```

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 !

Fiche d'identité de l' algorithme de **RECHERCHE SÉQUENTIELLE**

Principe : on cherche v dans une liste , on ….....→ renvoie VRAI si v est trouvé, faux sinon

Exemple : feuilleter un livre page après page depuis le début pour retrouver une information

Pseudo Code

…….

……

……

……

……

Python

```
def chercher1(v,list):          #recherche la présence de l'entier v dans la liste "list"
    for i in list:              # on parcourt la liste
        if i==v:                # v est trouvé dans la liste
            return True         # on indique qu'on a trouvé l'élément
    return False                # la liste est parcourue : v ne lui appartient pas

def chercher2(v,list):          #recherche la présence de l'entier v dans la liste "list"
    if v in list:               # on parcourt la liste
        return True             # v est trouvé dans la liste et on indique qu'on a trouvé l'élément
    return False                # la liste est parcourue : v ne lui appartient pas
```

TERMINAISON de l'algorithme

Nous allons nous intéresser au fait que cet algorithme s'arrête....en effet, prouver ici que l'algorithme remplit bien son rôle (Correction de l'algorithme) est « trivial » (mais ce ne sera pas le cas des algorithmes de tri!)

Pour prouver qu'un algorithme s'arrête, on choisit une variable et on vérifie que la suite formée par les valeurs de cette variable au cours des itérations converge en un nombre fini d'étapes vers une valeur satisfaisant la condition d'arrêt. Cette variable s'appelle un **variant de boucle.**

Dans ce cas précis, on peut choisir la suite du nombre d'éléments $(n_i)_{i\in N^*}$ de la liste qui ….....

* Supposons que $n_1 = 0$; la liste est …....., on n'entre pas dans la boucle et ….....est retourné (l'algorithme ….....)

Si $n_1>0$ et vaut la longueur de la liste de départ : on entre dans la boucle, on compare v au 1er élément ; soit il est égal à v et on ….....de la boucle en renvoyant vrai(et l'algorithme .....) ; soit il n'est pas égal à v et n_2 = ….....

* A la k^{ième} itération, trois possibilités sont présentes :
  - soit v est égal au k^{ème} terme : on sort de la boucle en renvoyant ….....(et l'algorithme se termine)
  - soit v n'est pas égal au k^{ème} terme et $n_{k+1} = n_k - 1 = n_{k-1} - 2 = n_{k-2} - 3 = \ ... = n_1 - (k-1) = n_1 - k + 1$
  - soit $n_k = 0$, tous les éléments de la liste ont été comparés à v sans succès, on sort de la boucle et …..... est retourné (l'algorithme se termine donc)

* La suite strictement décroissante $(n_i)_{i\in\mathbb{N}^*}$ est minorée par …..... : elle converge donc (on décrémente à chaque tour ce nombre de une unité)

Complexité en temps : Dans le pire des cas (celui où v n'appartient pas à la liste) , on parcourt l'ensemble de la liste de longueur n *:* on effectue n comparaisons, le temps de calcul de la fonction EST PROPORTIONNEL à la longueur de la liste. On parle d'une complexité en **O(n)** (lire « grand o de n ») : c'est un **coût linéaire**

Analyser l'algorithme suivant :

v ← élément cherché *#valeur du nombre entier cherché*

deb ← indice du premier élément de la liste

fin ← indice du dernier élément de la liste

r ← Faux *#Réponse initialisée à Faux*

i ← deb

Tant que i<= fin et que r=Faux faire

Si list[i] = v alors

r ← Vrai

FinSi

i ← i+1

FinTant que

Renvoyer la valeur de r

puis le faire tourner « à la main » sur la liste suivante comme dans l'exemple

`list=[19,2,81,70,7,97,85,26,45,86]` en recherchant 2 d'abord puis 77 ensuite.

Exemple avec 81 *_____v=81* *r=Faux* *deb=0 et fin= 9* *1 – list[0]= 19 ≠ v alors i=0+1 = 1* *2- list[1]= 2 ≠ v alors i=1+1 = 2* *3- list[2]= 81 = v alors r=Vrai et i=2+1 = 3* *4- on renvoie Vrai* ***A VOUS*** *_____v=2*	*_____v=77*

Complexité en temps

Combien de tours de boucles ont été nécessaires pour chercher 81 ? 2 ? et 77 ?

Afin d'estimer la complexité de cet algorithme, on se place dans le « pire des cas » , c'est à dire le plus grand nombre de tours de boucle possible : décrivez les conditions de cette situation pour un nombre v donné et une liste donnée…….

Dans ce pire des cas,combien d'étapes seront nécessaires pour chercher v dans une liste de longueur n?

Exercices

Exercice 1 – *CONCEVOIR ET TRADUIRE*

Écrire un algorithme qui compte le nombre d'éléments dans une liste, ce sans faire appel à une fonction dédiée (comme *len( )* en Python)
Indiquer la complexité temporelle si la liste contient n entiers.
Implémenter cet algorithme en Python

Exercice 2 – *CONCEVOIR ET TRADUIRE*

1°) Écrire un algorithme qui permette de trouver le minimum d'une liste d'entiers. Pour cela, on peut parcourir les éléments de la liste et utiliser mini, une variable initialisée au premier élément de la liste et n le nombre d 'éléments de la liste
Estimer le coût en temps d'un telle recherche
Implémenter cet algorithme en Python

2°) Améliorer cet algorithme en *minimum_1indice(liste)* pour qu'il renvoie aussi le 1er indice du minimum rencontré
A FAIRE ABSOLUMENT **Créer un algorithme *minimum_indice(liste,j)* qui sera utile pour le tri séquentiel : il prend en paramètres la liste et un rang pour renvoyer l'indice du du minimum de la liste comprise entre les indices j et n-1**

3°) Créer enfin un algorithme *minimum_indiceS(liste)* qui renvoie TOUS les indices de chacune des occurrences du minimum dans la liste. Estimer le coût en temps d'une telle recherche

Exercice 3 – *TRADUIRE ET DÉVELOPPER*

On associe des numéros à des personnages :

Personnage	Numéro
Celine	1
Lucas	2
Stephane	3
Louison	4
Jean	5

Ceci se traduit en Python par la liste :
Pers = [['Celine',1],['Lucas',2],['Stephane',3],['Louison',4],['Jean',5]]

Écrire une fonction qui permet de rechercher si un personnage de numéro v (un entier) quelconque est présent dans la liste : la fonction renverra le nom du personnage alors et *False* sinon

Exercice 4 – *ANALYSER ET CONCEVOIR*

On prétend que la fonction suivante teste l'appartenance de la valeur v au tableau t

```
def appartient(v,t):
    for i in range(len(t)):
        if t[i] == v:
            trouvee = True
        else :
            trouvee = False
    return trouvee
```

Donner des tests pour cette fonction et en particulier des tests montrant plusieurs raisons pour laquelle cette fonction est incorrecte

Que teste cette fonction en réalité ?
Aller plus loin....et que pensez vous de

(return a changé de place)

```
def appartient(v,t):
    for i in range(len(t)):
        if t[i] == v:
            trouvee = True
        else :
            trouvee = False
    return trouvee
```