



PARTIE 1 : Parcourir une liste.....	2
1. Algorithme de vérification .....	2
2. Algorithme de cumul.....	2
3. Recherche d'extremum.....	3
4. Complexité en temps .....	3
4.1. Pourquoi faire un calcul de complexité ?.....	3
4.2. Les deux types de complexité .....	4
4.3. Complexité en espace .....	4
4.4. Complexité en temps .....	4
4.5. Calcul de la complexité .....	4
4.5.1. Règles générales .....	4
4.5.2. Algorithmes sans structure de contrôle.....	5
4.5.3. Algorithmes avec structure conditionnelle.....	5
4.5.4. Algorithmes avec structure itérative .....	5
4.5.5. Algorithmes avec deux boucles itératives imbriquées .....	6
4.5.6. Ordre de grandeur .....	6

## **PARTIE 1 : Parcourir une liste**

### **1. Algorithme de vérification**

On parcourt une liste en faisant une recherche / vérification à chaque étape. On s'arrête dès que l'on trouve ce que l'on cherche ou un contre-exemple dans le cas d'une vérification.

Voici une fonction qui renvoie l'indice de la première personne qui a au moins 15 ans dans une liste et None si aucune personne ne correspond au critère.

```
# Renvoie l'indice de la première personne qui a au moins 15 ans
# dans une liste et None si aucune personne ne correspond au critère.
def premierMoins15ans(listePersonnes):
    for i in range(len(listePersonnes)):
        (nom ,age)=listePersonnes[i]
        if age <= 15:
            return i
    return None

assert premierMoins15ans([('Zoe' ,17),('Lea' ,19),('Leo' ,12),('Ano' ,14)]) == 2
assert premierMoins15ans([('Zoe' ,17),('Lea' ,19),('Dora' ,21),('Axel' ,18)]) == None
```

..

🔥 Exercice 1 : Créer une fonction vérifiant qu'une personne dont le nom est passé en paramètre est bien membre d'une liste de personnes passée également en paramètre. La valeur retournée sera un booléen.

```
listeNoms = ['Zoe','Lea','Leo','Ano']
assert verifNom('Zoe',listeNoms) == True
assert verifNom('Didier',listeNoms) == False
```

..

🔥 Exercice 2 : Créer une fonction cherchant le premier mot de 5 lettres et plus dans une liste de mots.

```
listeMots = ['aaa','aaaa','aa','aaaaa','aa','aaaaaaa']
assert chercheMot(listeMots) == 'aaaaa'
```

..

### **2. Algorithme de cumul**


🔥 Exercice 3 : Créer une fonction `moyenne (liste : list) ->int` qui renvoie la moyenne d'âge arrondie à l'entier inférieur d'une liste de personne

```
# Fonction calculant la moyenne d'âge d'une liste de personnes
def moyenne(listePersonne):

    return moyenneAge

assert moyenne([('Zoe' ,17),('Lea' ,19),('Leo' ,12)]) == 16
```

..


 Exercice 4 : Créer une fonction qui compte le nombre de personnes d'une liste qui ont strictement moins de 15 ans.

```
# Fonction renvoyant le nombre de personnes ayant strictement moins de 15 ans
def nbMoins15ans(listePersonnes):

    return compteur

assert nbMoins15ans([('Ano',14),('Lea',19),('Leo',12)]) == 2
```


..\scripts\part1\exo4.py

 Exercice 5 : Compter le nombre de voyelles dans une chaîne de caractère.

```
assert compteVoyelles('aaa') == 3
assert compteVoyelles('abc') == 1
assert compteVoyelles('') == 0
assert compteVoyelles('bcf') == 0
```

..\scripts\part1\exo5.py

### 3. Recherche d'extremum


 Exercice 6 : Créer une fonction renvoyant le nom de la personne la plus âgée d'une liste.

```
# Fonction renvoyant le nom de la personne la plus agee d une liste
def plusAgee( listePersonnes):

    return NomPlusVieux

assert plusAgee ([('Zoe',17),('Lea',19),('Leo',12)]) == 'Lea'
```

..\scripts\part1\exo6.py

 Exercice 7 : Trouver le nombre le plus proche de 0 dans une liste de nombres

```
assert plusProcheZero([5,3,1,4,2]) == 1
assert plusProcheZero([5,0,1,4,2]) == 0
assert plusProcheZero([5,3,-1,4,2]) == -1
```

..\scripts\part1\exo7.py

### 4. Complexité en temps

#### 4.1. Pourquoi faire un calcul de complexité ?

L'objectif premier d'un calcul de complexité algorithmique est de pouvoir **comparer l'efficacité d'algorithmes résolvant le même problème**. Dans une situation donnée, cela permet donc d'établir lequel des algorithmes disponibles est le plus optimal.

Pour faire cela, nous chercherons à estimer la quantité de ressources utilisée lors de l'exécution d'un algorithme. Les règles que nous utiliserons pour comparer et évaluer les algorithmes devront respecter certaines contraintes très naturelles. On requerra principalement qu'elles ne soient pas tributaires des qualités d'une machine ou d'un choix de technologie.

En particulier, cela signifiera que ces règles seront indépendantes des facteurs suivants :

- Du langage de programmation utilisé pour l'implémentation.
- Du processeur de l'ordinateur sur lequel sera exécuté le code.
- De l'éventuel compilateur employé.

Le premier réflexe qui consisterait à coder les algorithmes sur un ordinateur puis à comparer leurs durées d'exécution n'est donc pas le bon. Il ne vérifie en effet pas les contraintes précédentes et ne permet donc pas de juger leurs qualités intrinsèques.

Nous allons donc effectuer des calculs sur l'algorithme en lui-même, dans sa version "papier". Les résultats de ces calculs fourniront une estimation du temps d'exécution de l'algorithme, et de la taille mémoire occupée lors de son fonctionnement.

Le premier à avoir systématisé le fait de calculer la complexité d'un algorithme est Donald Knuth (1938-) dans sa série d'ouvrage "The Art of Computer Programming".

#### 4.2. Les deux types de complexité

On distinguera deux sortes de complexité, selon que l'on s'intéresse au **temps** d'exécution ou à l'**espace** mémoire occupé.

#### 4.3. Complexité en espace

La complexité en espace est quant à elle la taille de la mémoire nécessaire pour stocker les différentes structures de données utilisées lors de l'exécution de l'algorithme.

#### 4.4. Complexité en temps

Réaliser un calcul de complexité en temps revient à décompter le nombre d'opérations élémentaires (affectation, calcul arithmétique ou logique, comparaison...) effectuées par l'algorithme.

Pour rendre ce calcul réalisable, on émettra l'hypothèse que toutes les opérations élémentaires sont à égalité de coût. En pratique ce n'est pas tout à fait exact mais cette approximation est cependant raisonnable.

On pourra donc estimer que le temps d'exécution de l'algorithme est proportionnel au nombre d'opérations élémentaires.



#### **DÉFINITION 1 : Coût d'un algorithme**

Le coût d'un algorithme est l'ordre de grandeur du nombre d'opérations arithmétiques ou logiques que doit effectuer un algorithme pour résoudre le problème auquel il est destiné.

Cet ordre de grandeur dépend évidemment de la taille  $n$  des données en entrée.

On parlera de coût **linéaire** s'il est "d'ordre"  $n$ , de coût **quadratique** s'il est "d'ordre"  $n^2$ .

#### 4.5. Calcul de la complexité

##### 4.5.1. *Règles générales*

Pour calculer la complexité, nous allons devoir examiner chaque ligne de code et l'y attribuer un coût en temps.

Le coût ainsi obtenu n'aura pas d'unité, il s'agit d'un nombre d'opérations dont chacune aurait le même temps d'exécution : 1.

Les opérations qui vont devoir être comptabilisées sont :

- Les affectations comptent pour 1 unité de temps :  
 $a \leftarrow 2$
- Les comparaisons comptent pour 1 unité de temps :  
 $2 < 3$
- L'accès aux mémoires comptent pour une 1 unité de temps :  
Lire  $a$
- Chaque opération élémentaire compte pour une 1 unité de temps :  
 $3 + 2$

## ✓ EXEMPLE :

Déterminons le coût de la ligne de code suivante :  
`a=a+1`

$T(n)=1(\text{affectation})+1(\text{accès à la mémoire } a)+1(\text{addition})=3$

#### 4.5.2. Algorithmes sans structure de contrôle

 Exercice C1 : Déterminer la complexité  $T(n)$  de cette algorithmme écrit en python.

```
def conversion(n:float)->list:
    h=n//3600
    m=(n-3600*h)//60
    s=n%60
    return h,m,s
```

..\scripts\part1\exo C1.py

On ne comptera pas la ligne 1 et 5. Dans la suite quand on s'intéressera à la complexité d'un algo écrit en Python nous ne prêterons pas attention à la ligne de définition de la fonction et au return.

#### 4.5.3. Algorithmes avec structure conditionnelle


 Exercice C2 : La fonction suivante calcule  $(-1)^n$  sans effectuer de produit mais en utilisant un test avec une alternative :

```
def puissanceMoinsUn(n:int)->int:
    if n%2==0:
        res=1
    else:
        res=-1
    return res
```

..\scripts\part1\exo C2.py

Déterminer la complexité  $T(n)$  de cet algorithme

#### 4.5.4. Algorithmes avec structure itérative

 Exercice C3 : Cette fonction utilise une structure for pour calculer la somme des n premiers entiers

```
def sommeEntiers(n):
    somme = 0
    for i in range(n+1):
        somme += i
    return somme
```

..\scripts\part1\exo C3.py

Déterminer la complexité  $T(n)$  de cet algorithme.

#### 4.5.5. Algorithmes avec deux boucles itératives imbriquées

##### 🔥 Exercice C4 :

```
def sommeElementsMatrice(mat):
    """ La fonction calcul la somme des éléments de la matrice carrée """
    somme = 0
    for i in range(len(mat)):
        for j in range(len(mat)):
            somme = somme + mat[i][j]
    return somme
```

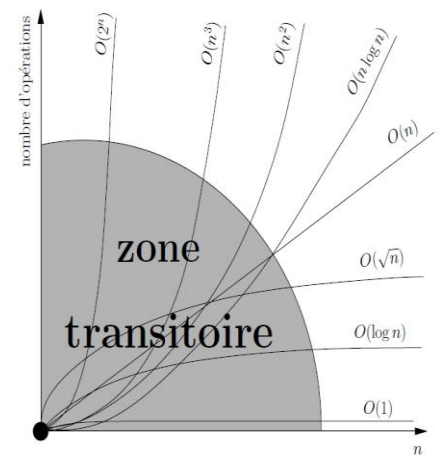
..\scripts\part1\exo C4.py

Déterminer le complexité  $T(n)$  de cet algorithme.

#### 4.5.6. Ordre de grandeur

Il n'est pas question de calculer exactement le nombre d'opérations engendrées par l'application d'un algorithme. On cherche seulement un ordre de grandeur (noté grand O).

L'ordre de grandeur asymptotique est l'ordre de grandeur lorsque la taille des données devient très grande.



##### ✓ EXEMPLE :

DÉBUT	
ÉCRIRE "Quelle est la taille du tableau, n : "	1 fois
LIRE n	1 fois
POUR i = 1 JUSQU'À n INCRÉMENT 1 FAIRE	n fois
SI (3*i^2 + i - 2) % 5 = 0 ALORS	n fois
T[i] <- 1	
SINON	n fois
T[i] <- 0	
FINSI	
FINPOUR	
POUR i = 1 JUSQU'À n INCRÉMENT 1 FAIRE	n fois
ÉCRIRE "Tableau[" , i , "]" = " , T[i]	n fois
FINPOUR	
FIN	

Au total nous avons donc un nombre total d'opérations élémentaires qui est égal à :

$$1 + 1 + n + n + n + n + n + n = 5n + 2 = O(n).$$

La relation  $O(n) = 5n + 2$  est « dominée asymptotiquement par  $n$  ». C'est-à-dire que si  $n$  est suffisamment grand, il existe une constante  $K$  qui permettra d'avoir  $K.n \geq 5n + 2$ .

Et par conséquent, la complexité de notre algorithme " d'initialisation d'un tableau d'entiers " est  $O(n)$ . On dit aussi que cet algorithme est **linéaire**.

Pour trouver l'ordre de grandeur  $O$  à partir du nombre d'opérations élémentaires, il faut :

- supprimer la constante
- garder uniquement le  $n$  qui possède l'exposant le plus grand
- supprimer le coefficient devant le  $n$