CHAPITRE 2 : COMPLEXITÉ ALGORITHMIQUE

I. Introduction :

Il existe souvent plusieurs façons de programmer un algorithme. Si le nombre d'opérations à effectuer est peu important et les données d'entrée de l'algorithme sont de faibles tailles, le choix de la solution importe peu. **En revanche**, lorsque le nombre d'opérations et la taille des données d'entrée deviennent importants, deux paramètres deviennent déterminants : le temps d'exécution et l'occupation mémoire.
On n'exige donc pas seulement d'un algorithme qu'il résolve un problème, on veut également qu'il soit efficace, c'est-à-dire :

- rapide (en termes de temps d'exécution),
- peu gourmand en ressources (espace de stockage, mémoire utilisée).

On a alors besoin d'outils qui nous permettront d'évaluer la **qualité théorique** des algorithmes proposés. Le coût de la mémoire étant aujourd'hui relativement faible, on cherche en général à améliorer la complexité en temps plutôt que la complexité en mémoire. Pour cette raison, on s'intéressera donc au **temps d'exécution**.

II. Notion de complexité

II-1. Définition:

- **La complexité d'un algorithme en temps :** donne le nombre d'opérations effectuées lors de l'exécution d'un programme.
  On appelle C_i le coût en temps d'une opération i.
- **La complexité en mémoire** : donne le nombre d'emplacements mémoires occupés lors de l'exécution d'un programme.

II-2. Intérêt de la complexité :

La complexité permet de :

- classer les problèmes selon leur difficulté,
- classer les algorithmes selon leur efficacité,
- comparer les algorithmes correspondant à un problème donné sans avoir à les implémenter, c'est-à dire à les traduire dans un langage particulier.

II-3. Complexité dans les différents cas :

On distingue la complexité dans le **pire des cas**, la complexité dans le **meilleur des cas**, et la complexité en **moyenne**. En effet, pour un même algorithme, suivant les données à manipuler, le résultat sera déterminé plus ou moins rapidement.

- **La complexité au meilleur** : est le plus petit nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille fixée. C'est une borne inférieure de la complexité de l'algorithme sur un jeu de données de taille n.

- **La complexité au pire :** est le plus grand nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille fixée. Comme il s'agit d'un maximum, l'algorithme finira donc toujours avant d'avoir effectué ce nombre maximum d'opérations.
  Cette complexité peut cependant ne pas refléter le comportement " usuel " de l'algorithme, le pire cas ne pouvant se produire que rarement.

- **La complexité en moyenne :** est plus difficile à calculer. Il ne s'agit pas comme on pourrait le penser de la moyenne des complexités au mieux et au pire. Concrètement, on se donne la probabilité d'apparition de chacun des jeux de données.
  Cette complexité en moyenne reflète le comportement " général " de l'algorithme si les cas extrêmes sont rares ou si la complexité varie peu en fonction des données.

✍ **G**énéralement, on s'intéresse au cas le plus **défavorable**, à savoir, la complexité dans le **pire des cas**.

II-4. La complexité asymptotique : Notation Grand-O :

La complexité asymptotique d'un algorithme décrit le comportement de celui-ci quand la taille n des données du problème traité devient de plus en plus grande (tend ver l'infini), plutôt qu'une mesure exacte du temps d'exécution.
La notation **O(f)** décrit le comportement asymptotique d'une fonction **f**, c'est à dire pour des grandes valeurs.

II-5. Classes de complexités :

Les complexités d'algorithmes (dans le pire des cas, que l'on notera O) les plus courantes sont :

Complexité	temps d'exécution	Description
O(1)	**temps constant**	Le temps d'exécution ne dépend pas des données traitées, ce qui est assez rare !
O(log(n))	**logarithmique**	Réponse immédiate **exemple : recherche dichotomique**
O(n)	**linéaire**	augmentation linéaire du temps d'exécution quand le paramètre croit (si le n double, le temps double). **exemple : recherche séquentielle**
O(n log(n))	**quasi-linéaire**	augmentation un peu supérieure à O(n) **exemple : Tri par fusion**
$O(n^2)$	**quadratique**	algorithmes avec deux boucles imbriquées. : **Exemple : Tris standards (Sélection, Bulle, Insertion)**
$O(n^p)$	**polynomiales**	ici, n^p est le terme de plus haut degré d'un polynôme en n ; il n'est pas rare de voir des complexités en $O(n^3)$ ou $O(n^4)$.
$O(p^n)$	**exponentielle**	quand le paramètre double, le temps d'exécution est élevé à la puissance p avec p > 1.

✍ Un algorithme logarithmique est considérablement plus efficace qu'un algorithme linéaire (lui-même plus efficace qu'un algorithme quadratique ou pire exponentiel). Cette différence devient

fondamentale si la taille des données est importante : le résultat peut se compter en millisecondes pour une méthode et en heures pour une autre !

Exemple :

Avec un ordinateur qui fait 10^9 instructions par seconde, on aurait les temps d'exécution :

Complexité	temps d'éxécution en fonction du nombre d'opérations					
n	5	10	15	20	100	1000
$\log n$	3.10^{-9} s	4.10^{-9} s	4.10^{-9} s	5.10^{-9} s	7.10^{-9} s	10^{-8} s
$2n$	10^{-8} s	2.10^{-8} s	3.10^{-8} s	4.10^{-8} s	2.10^{-7} s	2.10^{-6} s
$n \log n$	$1{,}2.10^{-8}$ s	3.10^{-8} s	6.10^{-8} s	10^{-7} s	7.10^{-7} s	10^{-5} s
n^2	$2{,}5.10^{-8}$ s	10^{-7} s	$2{,}3.10^{-7}$ s	4.10^{-7} s	10^{-5} s	10^{-3} s
n^5	3.10^{-6} s	10^{-4} s	$7{,}6.10^{-4}$ s	3.10^{-3} s	10 s	10^6 s 11 jours
2^n	$3{,}2.10^{-8}$ s	10^{-6} s	$3{,}3.10^{-5}$ s	10^{-3} s	$1{,}2.10^{21}$ s 4.10^{13} ans	10^{292} s 3.10^{284} ans
$n!$	$1{,}2.10^{-7}$ s	4.10^{-3} s	$1{,}4.10^3$ s 23 minutes	$2{,}4.10^9$ s 77 ans	10^{147} s 3.10^{141} ans	10^{500} s
n^n	3.10^{-6} s	10 s	$4{,}4.10^8$ s 13 ans	10^{17} s 3.10^9 ans	10^{191} s 3.10^{183} ans	10^{300} s

III. Calcul de la complexité :

L'efficacité d'un algorithme ne se mesure pas en secondes, par exemple, car cela impliquerait justement de les implémenter mais de plus, ces mesures ne seraient pas significatives car dépendantes de la machine utilisée.
On utilise donc des unités de temps abstraites correspondant au nombre d'opérations effectuées. Chaque opération (addition, multiplication, comparaison, incrémentation, affectation, ...) consomme alors une unité de temps. Le nombre d'opérations est égal à la somme de toutes les opérations.

III-1. Règles de calcul du grand O

- Si le coût d'un algorithme est une valeur constante ⟶ **O(1)**
  Exemple :
  ***C(algorithme)=* 15** alors la complexité asymptotique est **O(1)**

- Si le coût d'un algorithme dépend de n : on prend le plus grand degré
  Exemple :
  ***C(algorithme)=* $5n^2 + 2n+13$** alors la complexité asymptotique est **$O(n^2)$**

a. **Le coût des instructions élémentaires :** Le coût d'une opération élémentaire = 1

On appelle opération de base, ou opération élémentaire, toute :

- Affectation : x=2
- Test de comparaison : ==; <; <=; > ; >=; ! =
- Opération de lecture (input) et d'écriture (print)
- Opération arithmétique : + ; - ; * ; / ; // ; ** ; %;

Exemple : Que vaut le coût de l'algorithme A

```
x=2    #instr1    ⟶ C(instr1)=1
x=x*2  # instr2   ⟶ C(instr2)=2
print(x) # instr3 ⟶ C(instr3)=1
```

C(A)= C(instr1)+ C(instr2)+ C(instr3)=4 ⟶ O(1)

b. Le coût de if :

```
if condition:              # la condition est en c1
        instruction_1      # en c2
else:
        instruction_2      # en c3
```

- le if sera en **max(c1+c2,c1+3)**
- Cette règle peut être généralisée pour une instruction if ayant une ou des parties elif.

Exemple : Que vaut la complexité de cet algorithme :

```
if i%2==0 :        # 2
        n=i//2     #2
else :
        i=i+1      }#4
        n=i//2     }
```

La complexité de l'algorithme est de : ***O(1)***

c. Le coût des boucles (for ou while):

Le temps d'une boucle est égal à la multiplication de nombre de répétition par la somme du coût de chaque instruction x_i du corps de la boucle ;

- Coût(boucle for) =$\sum C\ (xi)$
- Coût (boucle while) =$\sum(C(comparaison) + C(xi))$

Exemple 1: Que vaut la complexité de cet algorithme :

```
for i in range(6):    #l nombre de répétition = 6
        print(i)      #le coût est 1
```

La complexité de l'algorithme est de $\sum_0^5 1$ ***=6 ⟶ O(1)***

Exemple 2: Que vaut la complexité de cet algorithme :

```
for i in range(n):    #l nombre de répétition = n
```

```
print(i)        #le coût est 1
```

La complexité de l'algorithme est de $\sum_{i=0}^{i=n-1} 1 = n \longrightarrow O(n)$

***Exemple** 3:* Que vaut la complexité de cet algorithme :

```
for i in range(n):      #l nombre de répétition = n
        for j in range(n-3):    #l nombre de répétition = n-3
                print(i+j)      #le coût est 2
```

La complexité de l'algorithme est de :

$\sum_{i=0}^{i=n-1} \sum_{j=0}^{j=n-4} 2 = \sum_{i=0}^{i=n-1} 2(n-3) = 2n*(n-3) = 2n^2-6n \longrightarrow O(n^2)$

d. L'appel d'une fonction :

Lorsqu'une fonction est appelée, le coût de cette fonction est le nombre total d'opérations élémentaires engendrées par l'appel de cette fonction.

Exemple : Que vaut la complexité de cet algorithme :

```
def somme(n) :
        s=0                     #le coût est 1
        for i in range(n+1) : #le nombre de répétition = n+1
                s+=I            #le coût est 2
        return s                #le coût est 1


y=0                             #le coût est 1
for i in range(n):              #le nombre de répétition = n
        y+= somme(i)            #le coût est 2+ coût (somme(i))=2+(2i+4)
print(y)                        #le coût est 1
```

La complexité de la fonction somme est de : $1+(\sum_{i=0}^{i=n} 2)+1=1+2(n+1)+1=2n+4 \longrightarrow O(n)$

La complexité de l'algorithme est :$1+(\sum_{i=0}^{i=n-1} (2i+6))+1 \longrightarrow O(n^2)$

e. L'appel d'une fonction récursive:

Pour les fonctions récursives : le temps de calcul est exprimé comme une relation de récurrence (on compte le nombre d'appels récursifs).

***Exemple** 1:* Déterminez la complexité de la fonction récursive suivante

```
def factoriel(n) :
        if (n == 0) : return 1
        else : return n*factoriel(n-1)
```

Analyse de la complexité : $C(n)=C(n-1)+1$ avec $C(0)=1$

✍ Pour les problèmes de type :**Diviser pour régner** on applique le théorème Master theorem suivant :

Théorème : Master theorem

On considère un problème de taille n, qu'on découpe en a sous-problèmes de taille n/b permettant de résoudre le problème. Le coût de l'algorithme est alors :

C(1) = 1
C(n) = a ×C(n/b) + f(n)
Avec **f(n)= Reconstruction(n)= c×n$^{\alpha}$** en general
Et a≥1 et b>1

soit k=$\log_b$ a alors on distingue alors plusieurs cas :

- **Cas1 :** Si **O(**f(n)**)= O(n^k)**, alors **C(n) = O(n^k log(n))**.
- **Cas2 :** Si **O(**f(n)**)= O(n^{k-e})** pour une constante e>0, alors **C(n) = O(n^k).**
- **Cas3 :** Si **O(**f(n)**)=O(n^{k+e})** pour une constante e>0, et si af(n/b) < cf(n) pour une constante c<1 alors **C(n) = O(f(n))**

Exemple :

C(n)=4C(n/2)+n alors :

- f(n)=n
- k= **$\log_2$4=2**
- **O(f(n))=O(n^1)=O(n^{k-1})**

Donc la complexité : **O(n^2)**