Chapitre 1 : Complexité

1 Introduction

Un algorithme effectue un traitement sur des données. S'il produit le résultat escompté, on dit que l'algorithme est *correct*. On veut que l'algorithme soit *efficace*, c'est-à-dire que le *temps de calcul* du résultat soit raisonnable.

2 Plusieurs chemins mènent à Rome, mais...

Exemple : On veut déplacer le premier élément d'un tableau pour le mettre à la fin. C'est ce qu'on appelle un *décalage circulaire*.

Tableau de départ	Après le *décalage circulaire*
`[1, 2, 3, 4, 5]`	`[2, 3, 4, 5, 1]`

Première solution	Deuxième solution
1. Échanger les 1er et 2e éléments,	1. Sauvegarder le premier élément du tableau dans une variable,
2. échanger les 2e et 3e éléments,	2. décaler tous les éléments d'une case vers la gauche,
3. échanger les 3e et 4e éléments,	3. écrire l'élément sauvegardé en dernière position.
4. et ainsi de suite jusqu'au dernier élément.	

Les deux approches pour résoudre le cahier des charges sont correctes ; cependant, dans le cadre de ce cours , la question que nous nous posons est : *"Lequel est le plus rapide ?"*

3 Mesure des performances avec `time`

Pour répondre à la question précédente nous allons mesurer le temps d'exécution de chaque programme pour déterminer lequel est le plus rapide. La bibliothèque `time` fournit un chronomètre qui permet de mesurer le temps de calcul.

```
from time import time
# ...

tic = time()  # top départ
```

```
# ######################################################
#              Traitement à chronométrer
# ######################################################
tac = time()  # arrêt du chronomètre

print(round(1000 * (tac - tic), 2), " ms")  # En ms arrondi au centième
```

On peut calculer la moyenne d'un certain nombre de mesures des temps d'exécution pour limiter l'impact des fluctuations des temps d'exécution mesurés (plusieurs processus peuvent mobiliser le processeur par exemple, ...)

4 Plusieurs types de complexité

Les algorithmes peuvent être classés en fonction de leur complexité. Voici deux classes importantes.

Algorithmes de complexité linéaire Les algorithmes qui comportent une *boucle simple* un nombre constant C d'opérations à chaque itération (pas de boucles imbriquées ou d'appels de fonction), font au total Cn opérations. Le temps de calcul est donc proportionnel à n, autrement dit, dépend **linéairement de** n. On dit qu'ils ont une **complexité linéaire**, ou une complexité en $O(n)$, prononcé *grand Ô de n*.

Note : Une opération est une "*opération simple*" arithmétique, logique, d'affectation etc.

Algorithmes de complexité quadratique Il s'agit par exemple d'algorithmes ayant une boucle imbriquée dans laquelle le nombre d'opérations est proportionnel à `n` et qui est exécutée un nombre de fois proprtionnel à `n`, comme une fonction qui calcule les tables de multiplication jusqu'à `n * n` et les écrit dans un dictionnaire `t`.

5 Complexité dans le pire des cas

En général, le nombre d'opérations dépend non seulement de la taille des données, mais des données elles-mêmes (on dit aussi : de l'instance).

Par exemple, si on recherche un élément particulier dans un tableau de taille `n`. Avec un jeu de données particulièrement heureux, l'élément est trouvé du premier coup et le programme s'arrête après une opération.

Mais si on étudie le cas du jeu de données le plus défavorable ou de la **complexité dans le pire des cas**. Dans l'exemple de la recherche, le pire des cas se produit quand l'élément n'est pas trouvé ou trouvé en dernier, ce qui oblige à parcourir le tableau entier ; la complexité est donc $O(n)$ dans le pire cas.