CH 10 : COMPLEMENTS D'ALGORITHMIQUE

VALIDITE, COMPLEXITE

Travail à faire : Activité 1

I. VALIDITE D'UN ALGORITHME

Lorsqu'on écrit un algorithme, il est important de vérifier que celui-ci fournira, en un temps fini, le résultat voulu. On dit alors que l'algorithme est valide.

1. Terminaison

La première question à se poser est de savoir si l'algorithme « se termine ». Le problème est notamment posé lorsqu'on utilise une boucle « WHILE » (dans le cas d'une boucle « FOR », le nombre d'étapes est défini au préalable, donc forcément fini).

Pour prouver la terminaison, on utilise un ***variant de boucle*** :

On choisit une expression, ou plus généralement une variable, dont la valeur est modifiée à chaque boucle. La suite formée par les valeurs de cette variable doit converger vers une valeur qui satisfait la condition d'arrêt.

2. Correction

La seconde question à se poser est : « l'algorithme fournit-il bien la réponse attendue ? ». Pour y répondre, on utilise cette fois un ***invariant de boucle***.

Un invariant de boucle est une propriété qui est vraie avant l'entrée dans la boucle, qui reste vraie à chaque passage dans cette boucle, et à la sortie de la boucle, où il doit correspondre au résultat attendu.

On utilise un raisonnement proche du raisonnement par récurrence :

- On démontre que la propriété est vraie avant l'entrée dans la boucle
- On vérifie que si la propriété est vraie avant un passage dans la boucle, elle est alors aussi vraie après le passage dans la boucle.
- On peut alors en conclure qu'elle est vraie à la sortie de la boucle.

II. COUT D'UN ALGORITHME

On doit également, lorsqu'on écrit un algorithme, se poser la question de son efficacité.

Si un programme doit traiter une liste de 10^7 éléments, puis une liste de 10^8 éléments, son temps d'exécution sera-t-il multiplié par 10 ? Quel est le rapport entre la taille de la liste et le temps d'exécution du programme ?

Les réponses sont diverses, et dépendent à la fois de l'algorithme et de la liste. Pour une même liste, un programme peut être plus rapide qu'un autre, mais avec un autre type de liste, ce peut être l'inverse. Le langage et la machine utilisés auront également une influence sur le temps d'exécution.

Pour comparer deux algorithmes, on peut se concentrer sur le nombre d'opérations à effectuer, en évaluant un ordre de grandeur de ce nombre en fonction de la taille des données. On s'intéressera toujours au « pire des cas », c'est-à-dire le cas où la donnée de départ implique le nombre maximum d'opérations et/ou d'itérations.

On parle de coût, ou complexité, d'un algorithme.

1. Complexité linéaire

Soit *n* la taille de la donnée. Si le nombre d'opérations de l'algorithme peut s'écrire $an + b$, on dit que ***la complexité est linéaire, ou $O(n)$.***

C'est le cas pour une boucle for classique :

Dans l'algorithme ci-contre, on a *n* passages dans la boucle, avec une affectation à chaque passage.

Le total des affectations est donc $n + 1$ (en comptant l'affectation initiale) et le coût est linéaire.

```
def puissance1(x, n):
    """ Renvoie x^n, pour n entier """
    p = 1
    for i in range(1, n+1):
        p = p * x
    return p
```

Exemple : un parcours séquentiel de liste (c'est-à-dire que la liste est parcourue élément par élément) est en général de complexité linéaire (détermination du minimum, du maximum, calcul de la moyenne, recherche d'une occurrence …)

2. Complexité quadratique

Soit *n* la taille de la donnée. Si le nombre d'opérations de l'algorithme peut s'écrire $an^2 + bn + c$, on dit que ***la complexité est quadratique, ou $O(n^2)$.***

C'est le cas notamment dans la plupart des cas lorsqu'on a deux boucles imbriquées (parcourt d'une liste de listes par exemple).

Dans l'algorithme ci-contre, il y a deux affectations d'initialisation.

On effectue n boucles sur i, avec une affectation à chaque fois, soit n affectations.

On effectue ensuite n fois la boucle sur j, et ce pour les n boucles sur j, soit n² affectations.

```
x = 0
n = 100
for i in range(n):
    x = x + i
    for j in range(n):
        x = x + j
```

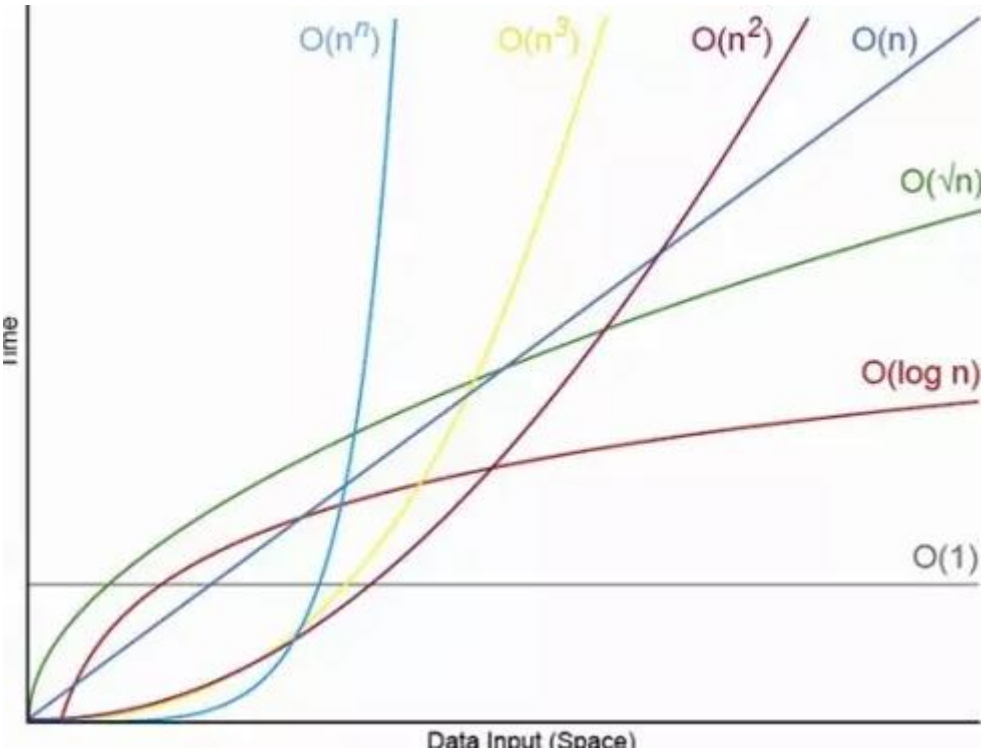
Au total, on aura donc effectué $n^2 + n + 2$ opérations, ce qui correspond à une complexité quadratique.

3. <u>Complexités</u>

Il existe bien sur d'autres types de complexités, mais dont l'étude théorique par « comptage » du nombre d'opérations est difficile. Les plus courantes sont :

$O(\log n)$	logarithmique
$O(n)$	linéaire
$O(n \log n)$	quasi-linéaire
$O(n^2)$	quadratique
$O(n^k) \quad (k \geqslant 2)$	polynomiale
$O(k^n) \quad (k > 1)$	exponentielle

On peut s'appuyer sur les courbes représentatives pour visualiser les différents types de complexité :



Il apparait clairement que certaines fonctions ont un coût qui augmente très vite, ce qui n'en fait pas des candidates intéressantes au niveau algorithmique ! Par contre, on essaiera, dans la mesure du possible, de se rapprocher d'un coût du type $O(Log(n))$ pour lequel on voit que la taille de l'objet à traiter fait peu évoluer le coût total.

Travail à faire : Activité 2

La recherche dichotomique sur une liste triée a un coût du type $O(Log(n))$, ce qui en fait un algorithme très efficace.