Complexité et efficacité d'un programme

Complexité d'un programme

Pour la plupart des programmes que nous avons pu faire, le temps d'exécution était en général de l'ordre de quelques millisecondes. Mais lorsqu'on augmente la taille des paramètres (taille des nombres ou longueur des tableaux), le temps de calcul peut augmenter. Même si nous n'allons pas chercher à calculer précisément le temps de calcul, nous allons essayer d'estimer son évolution en fonction de la taille des paramètres. Lorsqu'on multiplie par deux la taille des paramètres, est-ce que le temps d'exécution sera multiplié par 2, par 4, ou mis au carré ?
On appelle cela la **complexité**, ou le **coût**, d'un algorithme (ou programme). Il y a deux types de complexité. La **complexité spatiale** correspond à la place en mémoire occupée par le programme. Nous nous concentrerons plutôt sur la **complexité temporelle**, qui correspond au temps d'exécution du programme. Le temps "mesurable" dépend d'énormément de facteurs en dehors du programme lui même : rapidité de l'ordinateur, qualité de l'interpréteur Python utilisé, nombre de programmes tournant en parallèle sur le processeur... Puisque nous n'avons aucun contrôle sur ces aspects, nous nous concentrerons sur ceux liés à l'algorithmique, c'est-à-dire le nombre d'instructions qui seront exécutées par le programme. Étudier la complexité d'un programme n'a de sens que s'il contient des boucles, sous une forme ou une autre. Considérons les 3 programmes ci-dessous :

```
def prog1(n):
    s = 1
    return s
```

```
def prog2(n):
    s = 1
    for i in range(n):
        s = s + 1
    return s
```

```
def prog3(n):
    s = 1
    for i in range(n):
        for j in range(n):
            s = s + 1
    return s
```

On peut remarquer que la valeur de `s` correspond au nombre d'affectations effectuées lors de l'exécution des fonctions.

Exercice : Compléter le tableau ci-contre en mettant la valeur de `s` obtenue à partir de la valeur de `n` donnée.

`n`	`prog1(n)`	`prog2(n)`	`prog3(n)`
10			
20			
100			
n			

Pour les fonctions `prog2(n)` et `prog3(n)`, on peut remarquer que le "+1" n'a pas vraiment d'importance lorsque `n` est asssez grand.
Quand on compare les résultats obtenus pour `n` = 20 ou `n` = 100 par rapport à `n` = 10, on voit que l'évolution n'est pas du tout la même.

- Pour la fonction `prog1(n)` le nombre d'affectations ne dépend pas de `n`. On dit que la fonction a une complexité **constante**. On la note O(1). Le temps d'exécution sera sensiblement le même, quelque soit la valeur de `n`.
- Pour la fonction `prog2(n)` le nombre d'affectations est quasiment égal à `n`. Lorsqu'on multiplie `n` par 2 ou 10, le temps de calcul évolue de la même manière. On dit que la complexité est **linéaire**. On la note O(n).
- Pour la fonction `prog3(n)` le nombre d'affectations est environ `n`^{2}. Lorsqu'on multiplie `n` par 2 ou 10, le temps de calcul est multiplié par 4 ou 100. On dit que la complexité est **quadratique**. On la note O(n^2).

La notation $O(f(n))$ s'appelle la **notation de Landau**. Elle signifie que le temps d'exécution de la fonction évolue de la même manière que $f(n)$ lorsque n devient très grand. Dans le cas d'un polynôme, on ne garde que la plus grande puissance de n et on ne tient pas compte du coefficient multiplicateur. Par exemple, $5n^3 + 3n^2 - 7n + 1000$ correspond à $O(n^3)$.

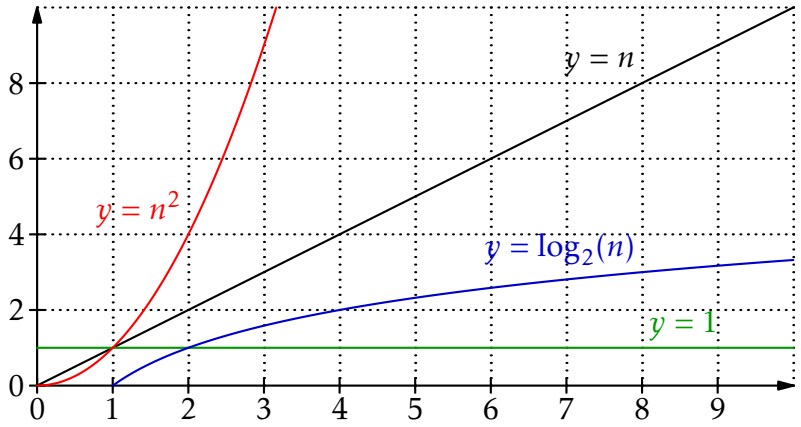
Complexité logarithmique

Il existe des classes de complexité qui se trouvent entre constante et linéaire. Pour illustrer cela, nous allons reprendre l'algorithme de multiplication à la russe.
Comme nous l'avons vu précédemment, le nombre d'itérations correspond au nombre de chiffres de l'écriture en binaire de a. Il est égal à l'entier k tel que :

$$2^{k-1} \leq a < 2^k$$

```
def multipl_russe(a, b):
    r = 0
    while a > 0:
        if (a%2 == 1):
            r = r + b
            a = a - 1
        a = a // 2
        b = b * 2
    return r
```

Il existe une fonction mathématique qui permet de calculer la valeur de k. On l'appelle le **logarithme en base 2** et on la note $\log_2(a)$, ou plus simplement, s'il n'y a pas d'ambiguïté, $\log(a)$. On parle donc de complexité **logarithmique**, noté $O(\log(n))$.
Lorsque la complexité est logarithmique, pour doubler le temps de calcul, il faut mettre au carré la taille des paramètres. Dans le cas de la multiplication, si $a = 1\,000$, il faudra prendre $a = 1\,000\,000$ pour doubler le temps de calcul.
On peut résumer l'évolution du temps de calcul en fonction de n pour les différents types de complexités à l'aide de la figure suivante.



Pour aller plus loin

De façon générale, on appelle les complexités du type $O(n)$, $O(n^2)$, $O(n^3)\ldots$ des complexités **polynomiales**. Avoir une complexité polynomiale peut sembler décevant, surtout si la puissance de n est grande, puisque cela veut dire que le temps de calcul augmente très vite. Mais il y a bien pire. Il y a les complexités **exponentielle**, notée $O(2^n)$ et **factorielle**, notée $O(n!)$. Lorsque n devient très grand, le temps de calcul devient gigantesque.
C'est justement la base de la plupart des algorithmes de cryptographie. Il est important que la complexité des fonctions permettant de décrypter un message sans connaître la clef secrète soit, au moins, exponentielle. Pour avoir un ordre de grandeur, les clefs de chiffrement sont choisies de telle sorte que même avec la puissance de tous les ordinateurs du monde, et le meilleur algorithme connu, la solution ne puisse pas être trouvée avant l'extinction du Soleil dans plus de 4 milliards d'années. Bien entendu, cela n'exclut pas qu'il existe un algorithme permettant de trouver le secret en moins de temps que cela.