Informatique — Complexité algorithmique

Ce n'est pas un hasard si l'informatique et la théorie du chaos se sont développées simultanément.

Faisons un bref retour en arrière (de quelques semaines).

On a construit dans un précédent TP un algorithme permettant **théoriquement** de déterminer si un entier est un nombre premier ou non ; mais l'exécution de cet algorithme peut demander un temps important (si l'entier est grand, et si on n'a pas trop de "chance").

En clair, à ce point du cours, on dispose d'un programme théorique parfait, marche "à tous les coups".

Mais cette affirmation ne tient pas compte de **deux aspects matériels cruciaux** : le **temps** et l'**espace**. Le temps tout d'abord, car il est clair que plus l'entier sera "grand" (ou plutôt, plus le premier diviseur de cet entier sera grand), plus la durée d'exécution sera grande. Et l'espace puisque les nombres intervenant dans ces calculs devront être stockés en mémoire, et occuperont de la place sur le disque dur.

En résumé :

> A ce stade, nous disposons d'un programme théorique parfait, qui donne en pratique la solution pour un nombre "pas trop gros", et donc sous réserve que l'exécution de ce programme ne demande pas trop de temps...

Ce qui affaiblit assez sensiblement l'enthousiasme initial. Alors, OK, prenons-en notre parti puisqu'il n'y a de toute façon pas d'alternative ; mais "pas trop gros", "pas trop de temps", qu'est-ce que cela signifie ?

La notion de **complexité** a pour objet de préciser ces questions. En fait, il faudrait plutôt parler de complexités car on distingue plusieurs types de complexité, dont voici l'échantillon le plus utile pour nos affaires :

- ☞ la **complexité algorithmique** est une estimation du nombre d'opérations élémentaires (au sens informatique) effectuée par un algorithme en fonction de la taille des données ;
- ☞ la **complexité temporelle** est une estimation de la durée d'exécution d'un algorithme en fonction de la taille des données ;
- ☞ la **complexité spatiale** est une estimation de l'espace mémoire nécessaire à l'exécution d'un algorithme.

Dans cette rapide présentation, nous nous restreindrons à l'étude de la complexité algorithmique. Pour expliciter cette quantité, il est nécessaire de savoir que les (principales) **opérations élémentaires** intervenant dans son calcul sont :

- ☞ les **opérations algébriques usuelles** : somme, différence, produit, quotient ;
- ☞ les **affectations** : le fait d'affecter à une variable une certaine valeur (attention : cas particulier de l'instruction `range`) ;
- ☞ les **comparaisons logiques** : $>$, $<$, $<=$, $>=$, $==$, `in`, `not in` ;
- ☞ une bonne partie des **fonctions usuelles** : `len`, `int`, `float`, `type`, `rand`...

Exemples de calculs pratiques de complexité

Dans la pratique, on notera souvent n la taille des données, et $C(n)$ la complexité algorithmique (le nombre d'opérations élémentaires) de l'algorithme.

Exemple 1 — Complexité algorithmique du calcul d'une moyenne.

Soient n un entier, et $L = [x_1, \dots, x_n]$ une liste de n réels. L'algorithme permettant de calculer la moyenne des valeurs de la liste L peut être :

```
1 MOY =0 # Initialisation de la moyenne
2 for k in range(n):
3         MOY =MOY +L[k]
4 MOY =MOY / len(L)
```

On détaille à présent le nombre d'opérations effectuées par l'algorithme ligne après ligne.

➤ La ligne 1 est une affectation : elle compte donc pour une opération élémentaire.

➤ La ligne 2 correspond à n affectations (on va donner à k successivement les valeurs 0, 1,..., $n-1$) : elle compte donc pour n opérations élémentaires.

➤ La ligne 3 (seule) correspond à 3 opérations : une extraction, une somme et une affectation. Comme elle sera exécutée n fois (au cours de la boucle), elle compte pour $3n$ opérations élémentaires.

➤ La ligne 4 correspond à 3 opérations : une somme, une affectation et un appel de la fonction len. Elle compte donc pour 3 opérations élémentaires.

Au final, le nombre d'opérations élémentaires $C(n)$ est ici : $\boxed{C(n) = 4n + 4}$.

Ainsi : $\boxed{C(n) = O(n)}$. On parle alors de **complexité algorithmique linéaire**.

Exemple 2 — Complexité algorithmique de la création d'une matrice nulle.

On considère ici un algorithme qui prend comme paramètre un entier n, et retourne la matrice carrée à n lignes et n colonnes nulle.

```
1 A =[] # 1 op. élém.
2 for j in range(n): # n op. élém.
3         A =A +[[]] # 2n op. élém.
4 for j in range(n): # n op. élém.
5         for k in range(p): # (n*n) op. élém.
6                 A[j] =A[j] +[0] # 2*(n*n) op. élém.
```

Au final, la complexité algorithmique $C(n)$ de cet algorithme est : $\boxed{C(n) = 3n^2 + 4n + 1}$.

Ainsi : $\boxed{C(n) = O(n^2)}$. On parle alors de **complexité algorithmique quadratique**.

Exemple 3 — Complexité algorithmique du calcul d'un plus grand élément d'une liste.

Soient n un entier, et $L = [x_1, \dots, x_n]$ une liste de n réels. L'algorithme permettant de déterminer le plus grand élément de la liste L peut être :

```
1 MAX =L[0]
2 for k in range(1,n):
3         if L[k] >MAX:
4                 MAX =L[k]
```

Ici se présente une situation nouvelle : la ligne 4 ne sera effectuée que si l'assertion `L[k] > MAX` est vraie. Doit-on alors la compter, ou pas, dans le calcul de la complexité ?

Une convention dans ce cas est de toujours envisager le pire, et de faire comme si l'assertion `L[k] > MAX` était vraie pour toute valeur de k. On parle d'ailleurs de **complexité au pire**. En clair, le détail du calcul du nombre d'opérations élémentaires consommées par l'algorithme ici est :

```
1 MAX =L[0] # 1 op. élém.
2 for k in range(1,n): # (n−1) op. élém.
3         if L[k] >MAX: # (n−1) op. élém.
4                 MAX =L[k] # (n−1) op. élém.
```

Au final, la complexité algorithmique (au pire) $C(n)$ de cet algorithme est : $\boxed{C(n) = 3n - 2}$; il s'agit encore d'une complexité linéaire.

Illustrations

> *Pour savoir combien de temps prend la réalisation d'un programme, faites votre estimation la plus fiable, multipliez par deux, et arrondissez à l'unité de temps supérieure. Ainsi un code que vous aviez prévu d'écrire en 2 minutes vous prendra effectivement quatre heures.*

En pratique, on commet souvent l'abus de confondre complexité algorithmique et complexité temporelle. L'idée est qu'une opération élémentaire demande sensiblement le même temps, qu'il s'agisse d'une addition, d'une affectation ou autre : **on fait l'approximation que chaque opération élémentaire demande la même durée pour être exécutée**. D'ailleurs, c'est le seul moyen raisonnable pour pouvoir faire le lien entre les deux complexités (algorithmique et temporelle) facilement.

Sous cette hypothèse, la complexité algorithmique $C(n)$ et la durée $T(n)$ d'un algorithme seront donc proportionnelles, dans un rapport essentiellement déterminé par la vitesse de votre processeur. Pour être plus explicite, imaginons (pour simplifier) que vous disposez d'une machine équipée d'un processeur ayant une fréquence d'un giga-hertz ; cela signifie qu'un milliard d'opérations pourront être réalisées chaque seconde.

En fonction de la complexité algorithmique d'un programme, on est alors capable d'estimer la durée nécessaire à son exécution.

Le tableau ci-dessous donne quelques exemples de telles estimations.

Ordre de grandeur de la complexité	Nature de la complexité	Ordre de grandeur de la durée (pour $n = 10^6$)	Exemple d'algorithme
$O(1)$	Constante	1 *ns*	Extraction d'un élément dans un tableau
$O(\ln(n))$	Logarithmique	15 *ns*	Algorithme de dichotomie
$O(\sqrt{n})$	Racinaire	1 μs	Test de primalité
$O(n)$	Linéaire	1 *ms*	Calcul de la moyenne
$O(n^2)$	Quadratique	15 *min*	Création d'une matrice carrée nulle
$O(n^3)$	Cubique	30 *ans*	Multiplication de 2 matrices $n \times n$
$O\left(2^{\text{polynôme}(n)}\right)$	Exponentielle	$10^{300\,000}$ *ans*	Algorithme du sac à dos
$O(n!)$	Factorielle	$10^{5\,000\,000}$ *ans*	Algorithme du voyageur de commerce

Exercices

Il y a trois types de programmes : ceux avec des bugs que vous connaissez, ceux avec des bugs que vous ne connaissez pas, et ceux avec les deux.

Exercice 1. — Calculer la complexité algorithmique d'un algorithme extrayant l'initiale d'un mot.

Exercice 2. — Calculer la complexité algorithmique d'un algorithme comptant le nombre de voyelles d'un mot.

Exercice 3. — Calculer la complexité algorithmique d'un algorithme calculant la trace d'une matrice.

Exercice 4. — Calculer la complexité algorithmique d'un algorithme du calcul de la somme de deux matrices (carrées de même taille).

Exercice 5. — Calculer la complexité algorithmique d'un algorithme du calcul du produit de deux matrices (carrées de même taille).

Exercice 6. — Calculer la complexité algorithmique d'un algorithme déterminant si un entier supérieur à 4 est de Carmichael ou non.