Chapitre 5 *Complexité*

La théorie de la complexité est le domaine des mathématiques, et plus précisément de l'informatique théorique, qui étudie formellement la quantité de ressources (temps, espace mémoire, etc.) dont a besoin un algorithme pour résoudre un problème.

L'objectif premier d'un calcul de complexité algorithmique est de pouvoir comparer l'efficacité d'algorithmes résolvant le même problème. Si nous devons par exemple trier une liste de nombres, est-il préférable d'utiliser un tri fusion ou un tri à bulles ?

Ce type de question est primordial, car pour des données volumineuses, la différence entre les durées d'exécution de deux algorithmes ayant la même finalité peut être de l'ordre de plusieurs jours.

1 Un premier exemple : le calcul de x^n pour $n \geq 1$

On souhaite calculer x^n à l'aide de deux algorithmes de complexité très différente.

1.1 Premier algorithme

La première idée consiste à remplir un tableau de taille n à l'aide des valeurs succcessives de x^n prises pour n=1, n=2, ...

Voici l'algorithme écrit en pseudo-code :

```
AlgoTableau (x, n)
T un tableau de taille n;
T[0] ⟵ x;
pour tous les i de 1 à n − 1 faire
    T[i] ⟵ x ∗ T[i − 1];
retourner T[n − 1];
```

Nous devons prouver la correction de l'algorithme (qui fera l'objet d'un prochain chapitre) :

- Terminaison
- Validité
- Complexité en espace

- Complexité en temps

1.2 Deuxième algorithme

La deuxième idée consiste à compter le problème en deux sous-partie, selon le principe du "Diviser pour régner" et ce, récursivement :

Voici l'algorithme écrit en pseudo-code :

```
ALGOD&C(x, n)
si n = 1 alors retourner x;
sinon
    z = ALGOD&C(x, ⌊n/2⌋);
    si n est pair alors retourner z × z;
    si n est impair alors retourner x × z × z;
```

- Terminaison
- Complexité en espace et en temps

- Nombre d'appels récursifs

Complexité (en espace et en temps)

1.3 Premières conclusions

2 Un modèle pour la complexité

Dans ce cours, on ne va s'intéresser qu'à la complexité en temps. Ce modèle de calcul de complexité va permettre de faire des prédictions sur le temps d'exécution d'un algorithme.

On distinguera :

- Les opérations élémentaires : déclaration de variable, opération arithmétique (+ - / X) les affectations, lecture, écriture de variable, test élémentaire, appel de fonction. Chaque opération élémentaire prendra un temps constant (on appelle ce modèle le modèle WORD-RAM).
- Les branchements : *si alors sinon* . Le coût d'une instruction conditionnelle **if b : p else : q** est inférieur ou égal au maximum des coûts des instructions p et q, plus le coût de l'expression booléenne b,

- Les boucles : *pour* et *tant que*. Le coût d'une boucle **for i in iterable : p** est égal au nombre d'éléments de l'itérable multiplié par le coût de p, si p ne dépend pas de i. Quand le coût du corps de la boucle dépend de i, le coût total de la boucle est la somme des coûts du corps de la boucle pour chaque valeur de i. Le cas des boucles **while** est plus complexe à traiter puisque le nombre de répétitions n'est a priori pas connu. On peut le majorer par la somme des coûts de chaque comparaison additionné du coût du corps de la boucle,
- L'appel d'une fonction a un coût égal au nombre total d'opérations élémentaires engendrées par l'appel de cette fonction.

Pour déterminer la complexité en temps d'un algorithme, on doit compter le nombre d'opérations élémentaires, et exprimer ces valeurs en fonction des paramètres d'entrée de l'algorithme, souvent n. Dans le pire des cas, et si on n'arrive pas à compter exactement, on établira une borne supérieure sur ces valeurs. On va effectuer une estimation asymptotique, à une constante multiplicative près, qui sera cachée dans les O, mais qui en pratique peut avoir son importance pour distinguer deux algorithmes.

3 Outils mathématiques

3.1 La notation de Landau $O()$

Dans les années 1960 et au début des années 1970, alors qu'on en était à découvrir des algorithmes fondamentaux, on ne mesurait pas leur efficacité. On se contentait de dire, par exemple :« Cet algorithme se déroule en 6 secondes avec un tableau de 50000 entiers choisis au hasard en entrée, sur un ordinateur IBM 360/91 ». Une telle démarche rendait difficile la comparaison des algorithmes entre eux. La mesure publiée était dépendante du processeur utilisé, des temps d'accès à la mémoire vive et de masse, du langage de programmation et du compilateur utilisé, etc.

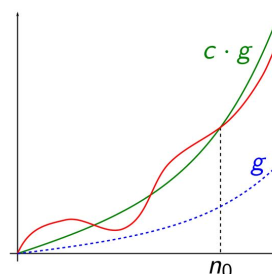
Comme il est très difficile de comparer les performances des ordinateurs, une approche indépendante des facteurs matériels était nécessaire pour évaluer l'efficacité des algorithmes. Donald Ervin Knuth (1938-) fut un des premiers à l'appliquer systématiquement dès les premiers volumes de sa fameuse série The Art of Computer Programming. La notation grand O (avec la lettre majuscule O) est un symbole utilisé en mathématiques, notamment en théorie de la complexité, pour décrire le comportement asymptotique des fonctions.

≪ Grand O ≫

Soit $f, g : \mathbb{N} \to \mathbb{R}_+$. Alors $\boldsymbol{f = O(g)}$ si

$$\exists c > 0, n_0 \geq 0, \forall n \geq n_0, f(n) \leq c \cdot g(n).$$

≪ f est un grand O de g s'il existe une constante c et un entier n_0 tels que pour toute valeur n plus grande que n_0, $f(n)$ est inférieur ou égal à $c \cdot g(n)$ ≫



En résumé : $f = O(g)$ si pour n suffisamment grand, f est plus petite que g, à une constante multiplicative près.

Exemple de cours :

1. Est-ce que $5n + 15$ est $O(n^2)$?
2. Est-ce que n^2 est $O(n^3)$?

Attention, $g(n)$ est $O(f(n))$ ne veut pas dire que $g(n)$ s'approche asymptotiquement de $f(n)$, mais seulement qu'au-delà de certaines valeurs de son argument, $g(n)$ est dominé par un certain multiple de $f(n)$.

3.2 Grand O et ses limites

Lemme
Soit $g : \mathbb{N} \to \mathbb{R}_+^*$ *(*$g(n) > 0$*). S'il existe une constante* $c \geq 0$ *telle que* $\lim_{n\to+\infty} \left(\frac{f(n)}{g(n)}\right) = c$*, alors* $f = O(g)$*.*

3.3 Calcul avec les grand O

- $O(f) + O(g) = O(f + g)$
- $O(f) \times O(g) = O(f \times g)$
- Si $f = O(g)$, alors $O(f + g) = O(g)$
- Si $\lambda \in \mathbb{R}_+$, $O(\lambda f) = O(f)$

3.4 Les fonctions du cours à connaître

- $log_2(n)$
- $\sqrt{n}$
- n
- n^2
- n^k
- 2^n
- $n!$

3.5 Lemme de croissance comparée

Pour tout $\alpha, \beta > 0$,

$$\lim_{+\infty} \frac{(\log n)^\alpha}{n^\beta} = 0 \qquad \lim_{+\infty} \frac{n^\alpha}{(2^n)^\beta} = 0 \qquad \lim_{+\infty} \frac{(2^n)^\beta}{n!} = 0$$

Si $\alpha < \beta$:

$$\lim_{+\infty} \frac{(\log n)^\alpha}{(\log n)^\beta} = 0 \qquad \lim_{+\infty} \frac{n^\alpha}{n^\beta} = 0 \qquad \lim_{+\infty} \frac{(2^n)^\alpha}{(2^n)^\beta} = 0$$

4 Ordre de grandeur des temps d'exécution

On considère ici un problème de taille 10^6 sur un ordinateur pouvant effectuer 10^9 opérations par seconde.

Complexité	Nom courant	Temps d'exécution	Remarque
$O(1)$	Temps constant	1ns	Le temps d'exécution ne dépend pas des données traitées, ce qui est assez rare
$O(log(n))$	Logarithmique	10 ns	Exécution quasi instantanée
$O(n)$	Linéaire	1 ms	Temps d'exécution minime
$O(nlog(n))$	Quasi-linéaire	1 ms	Temps d'exécution minime
$O(n^2)$	Quadratique	15 min	Cette complexité reste acceptable pour des données de taille moyenne ($n < 10^6$), mais pas au-delà
$O(n^3)$	Cubique	30 ans	
$O(n^k)$	Polynomiale	30 ans si k=3	Il n'est pas rare d'avoir des complexités en $O(n^3)$ ou $O(n^4)$
$O(2^n)$	Exponentielle	$> 10^{300000}$ milliards d'années	Très très long... dès que n>50
$O(n!)$	Factorielle	∞	Vraiment long ...

5 Différentes nuances de complexité

On peut distinguer plusieurs complexités différentes :

- Complexité au pire cas : c'est le plus grand nombre d'opérations qu'aura à exécuter un algorithme sur un jeu de données de taille fixée à n. C'est le seul cas à savoir traiter cette année,
- Complexité dans le meilleur des cas : elle n'est pas très utilisée mais elle peut donner une borne inférieure du temps d'exécution d'un algorithme,
- Complexité en moyenne : on peut l'évaluer si on a une idée de la fréquence des différentes données possibles pour un même problème de taille n. La théorie des probabilités et des statistiques doit intervenir,
- Complexité en espace : c'est la place en mémoire que requiert un algorithme pour fonctionner. Pour la déterminer, il suffit de faire le total des tailles en mémoire des différentes variables utilisées. Les fonctions récursives sont un exemple de complexité d'espace élevée, souvent cachée.
- Complexité amortie : c'est le cas lorsqu'un algorithme devant s'exécuter en $O(n)$, est en réalité en $O(1)$. C'est possible si une instruction interne à l'algorithme anticipe un besoin futur, ce qui permet d'améliorer globalement la vitesse d'exécution.

6 Complexité d'une pile (voir chapitre 4 et TD/compétence 7)

Les méthodes *push()*, *pop()*, *top()* et *isEmpty()* ont une complexité en temps de $O(1)$, c'est-à-dire que l'appel de ces méthodes a un coût identique quelle que soit la taille de la pile. C'est donc intéressant pour les piles de forte capacité. La complexité en mémoire est alors de l'ordre de $O(maxObjets)$.

* *
*

Version du 13 novembre 2019