Chapitre de révision : la complexité

Table des matières

1 Complexité temporelle des algorithmes

1) Quelques définitions

Un **algorithme** est une procédure de calcul bien définie qui prend en **entrée** une valeur, ou un ensemble de valeurs, pour produire en **sortie** d'autres valeurs ou ensemble de valeurs. Un algorithme est donc une séquence d'étapes de calcul qui transforment l'entrée en sortie.

En voici deux exemples :

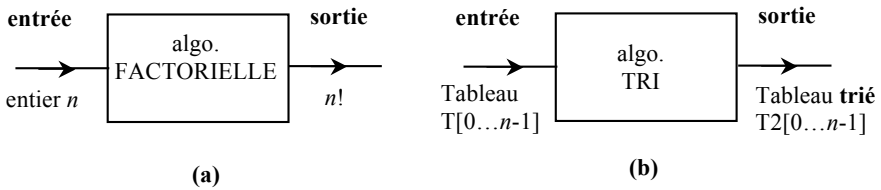

Figure 1 Algorithme vu comme une boîte possédant une entrée et une sortie. C'est une séquence d'étapes de calcul qui transforment l'entrée en sortie.

On appelle **taille de l'entrée** la "grosseur" caractéristique de l'entrée. C'est une notion un peu subjective, qui dépend surtout du type de calculs que fait l'algorithme. En gros, le nombre d'étapes de l'algorithme dépend directement de la taille de l'entrée :

- Dans le cas de l'algorithme FACTORIELLE, la taille de l'entrée est l'entier n lui même. Plus n est grand, plus le nombre d'étapes pour aboutir au résultat $n!$ est important.
- Dans le cas d'un algorithme de TRI d'un tableau T$[0...n-1]$, la taille de l'entrée est le nombre d'éléments à trier, c'est à dire n.

2) Complexité temporelle d'un algorithme

Cette notion est abordée dans le programme de MPSI. Lorsqu'on écrit un algorithme ou une fonction, il est important d'évaluer le temps qu'il prendra pour produire la sortie : on appelle cela sa **complexité temporelle**. Cette complexité temporelle est calculée en sommant les *coûts temporels*[1] de chaque étape, ce qui n'est rien d'autre que la durée de chaque étape.

Exemple 1

Prenons l'exemple de FACTORIELLE(n) dans l'approche itérative. Notons T(n) son temps d'exécution pour une entrée égale à n. Chaque

1. On préfère parler de coût temporel plutôt que de durée car on va voir plus tard qu'on peut définir d'autres types de coûts comme par exemple la taille de la mémoire de l'ordinateur nécessitée par une étape, ou encore le coût en opérations arithmétiques {+, -, * et / }, défini comme étant le nombre d'opérations arithmétiques nécessaires à chaque étape.

ligne L_i d'instruction a **un coût temporel** (durée) c_i et elle est **executée une ou plusieurs fois** :

	FACTORIELLE(n)	Coût temporel	Nbre de fois
1	**if** $n == 0$ **or** $n == 1$:	c_1	1
2	**return** n	c_2	1
3	**else** :	0	0
4	fact = 1	c_4	1
5	**for** i in range(2,n) :	c_5	$n-1$
6	fact = fact * i	c_6	$n-1$
7	**return** fact	c_7	1

Remarque :

L'instruction `else` ligne 3 ne coûte rien car elle n'est en réalité jamais exécutée. Le processeur évalue la vérité de l'expression `n == 0 or n == 1` à la ligne 1. Si elle est vraie (True) il passe à l'instruction de la ligne 2. Si elle est fausse (False) il passe directement à l'instruction de la ligne 4.

- $T(0) = T(1) = c_1 + c_2$.
- Pour $n > 1$, $T(n) = c_1 + c_4 + (n-1) \times c_5 + (n-1) \times c_6 + c_7$. En regroupant les termes, on obtient :

$$T(n) = (c_5 + c_6) \times n + (c_1 + c_4 - c_6 + c_7)$$

donc de la forme :

$$\boxed{T(n) = a \times n + b \quad \text{pour} \quad n > 1}$$

où a et b sont deux constantes.

Exemple 2 : (remarque [2])

Soit $n > 1$ un entier naturel. Étudions une fonction basée sur un algorithme qui calcule l'entier p vérifiant : $2^p \leqslant n < 2^{p+1}$.

précondition : $n \geqslant 1$

	fonction PUISS2(n) :	Coût temporel	Nbre de fois
1	p = -1	c_1	1
2	e = 1	c_2	1
3	**while** e <= n :	c_3	N
4	p = p + 1	c_4	N
5	e = 2*e	c_5	N
6	**return** p	c_6	1

Évaluons le nombre de fois N que l'algorithme parcourt la boucle while. Soit i le compteur de boucle ; avant l'entrée dans la boucle, $i = 0$ et :

- À la fin de la première boucle : $i = 1$ et $e(i) = 2^1$
- À la fin de la seconde boucle : $i = 2$ et $e(i) = 2^2$
- ...
- À la fin de la $N^{\text{ième}}$ boucle $i = N$ et $e(i) = 2^N$ avec

$$2^{N-1} \leqslant n < 2^N$$

2. On peut avoir une approche plus brutale avec les coûts temporels en convenant qu'ils sont tous égaux ($c_i = c$). Dans ce cas, le plus simple est de prendre $c = 1$, en convenant que ce " 1 " représente une **unité de temps-machine**, cette unité de temps pouvant valoir 1 μs, 1 ns par exemple (cela dépend de la machine utilisée).

Un outil indispensable : le logarithme en base 2

Le **logarithme en base 2**, noté lg est défini par :

$$y = \lg(x) \iff x = 2^y$$

On a immédiatement les propriétés suivantes (qui sont les propriétés classiques des logarithmes) :

- $\ln(x) = \ln(2) \times \lg(x)$. On remarque donc que $\lg(x)$ n'est défini que sur $\mathbb{R}_+^*$ et que c'est une fonction croissante [a].
- $\forall\,(x,y) \in \mathbb{R}_+^* \times \mathbb{R}_+^*,\ \lg(xy) = \lg(x) + \lg(y)$.
- $\forall\, a \in \mathbb{R}, \forall\, x \in \mathbb{R}_+^*\ \lg(x^a) = a\lg(x)$.
- $\forall\, a \in \mathbb{R},\ a = \lg(2^a)$. En particulier : $\lg(2) = 1$

a. $\ln(x)$ représente le logarithme népérien de x.

Il vient donc :

$$2^{N-1} \leqslant n < 2^N \iff N - 1 \leqslant \lg(n) < N$$

On constate facilement que :

$$T(n) = c_1 + c_2 + c_6 + N \times (c_3 + c_4 + c_5)$$

et comme $N \leqslant \lg(n) + 1$ il vient :

$$\mathrm{T}(n) \leqslant c_1 + c_2 + c_6 + c_3 + c_4 + c_5 + \lg(n) \times (c_3 + c_4 + c_5)$$

donc, de façon générale :

$$\boxed{T(n) \leqslant a \times \lg(n) + b}$$

où a et b sont deux constantes. On vera plus loin pourquoi il est important de majorer $\mathrm{T}(n)$.

Exemple 3 :

Prenons un dernier exemple avec la fonction FACTORIELLE(n) dans l'approche récursive :

	FACTORIELLE(n)	Coût temporel	Nbre de fois
1	**if** $n == 0$ **or** $n == 1$:	c_1	1
2	**return** n	c_2	1
3	**else** :	0	0
4	$x = n$ * FACTORIELLE($n-1$)	$c'_4 + \mathrm{T}(n-1)$	1
5	**return** x	c_5	1

Remarque :

Le coût temporel de la ligne 4 correspond au coût de FACTORIELLE($n-1$), c'est à dire $\mathrm{T}(n-1)$ plus le coût de la multiplication par n et de l'affectation à la variable x que nous avons noté c'_4.

On a donc :

- $\mathrm{T}(0) = \mathrm{T}(1) = \mathrm{c}_1 + \mathrm{c}_2 \overset{\text{noté}}{=} b$.
- Si $n > 1$, $\mathrm{T}(n) = \mathrm{c}_1 + \mathrm{c'}_4 + \mathrm{T}(n-1) + \mathrm{c}_5$. En regroupant les termes, nous obtenons : $\mathrm{T}(n) = \mathrm{T}(n-1) + \mathrm{c}_1 + \mathrm{c'}_4 + \mathrm{c}_5$, donc de la forme :

$$\mathrm{T}(n) = \mathrm{T}(n-1) + a$$

avec $a = \mathrm{c}_1 + \mathrm{c'}_4 + \mathrm{c}_5$.

$\mathrm{T}(n)$ est donc défini par une **récurrence** : il s'agit d'une suite arithmétique de raison c. On obtient finalement : $\mathrm{T}(n) = a \times n + \mathrm{T}(0)$. En conclusion, comme pour l'approche itérative, $\mathrm{T}(n)$ est une fonction affine de n. On a donc :

$$\boxed{\mathrm{T}(n) = a \times n + b}$$

3) Notation O

D'une façon générale, n représente la **taille** de l'entrée et $\mathrm{T}(n)$ le temps que met l'algorithme (ou la fonction) pour obtenir la sortie avec une entrée de taille n.

Ce qui importe le plus au programmeur est le **comportement asymptotique** de $\mathrm{T}(n)$ lorsque n est grand. En effet, ce sont les grandes valeurs de la taille de l'entrée qui qui auront l'effet le plus critique sur le temps de calcul (très énervant pour l'utilisateur qui doit patienter parfois assez longtemps...).

Pour rendre cela opérationnel et efficace, il est nécessaire d'introduire les définitions et notations suivantes :

Définition et notation O

Soient f et g deux applications de $\mathbb{N} \longrightarrow \mathbb{R}_+$, à *valeurs réelles positives*. On dit que $f = \mathrm{O}(g)$ (prononcer "f est grand O de g") si et seulement si :

$$\exists\, N \in \mathbb{N} \text{ et } \exists\, c \in \mathbb{R}_+ \text{ (constante) t.q. } \forall\, n \geqslant N,\ \ f(n) \leqslant c \times g(n)$$

On dit que f est **dominée** par g.

Règles de calcul avec O

f, g et h étant trois applications de $\mathbb{N}$ dans $\mathbb{R}_+$, on a :

1. $f = \mathrm{O}(f)$
2. Si $f = \mathrm{O}(g)$ et $g = \mathrm{O}(h)$, alors $f = \mathrm{O}(h)$.

   C'est la propriété de transitivité de la notation O.
3. Si $f = \mathrm{O}(g + h)$ et que $\lim\limits_{n \to +\infty} \dfrac{h(n)}{g(n)} = 0$, alors $f = \mathrm{O}(g)$

   Cela signifie que, dans une somme de deux termes, seul le terme prépondérant est important.
4. Si $a \in \mathbb{R}_+$ (constante positive) et si $f = \mathrm{O}(ag)$ alors $f = \mathrm{O}(g)$

   Une constante multiplicative peut être omise.

Preuves :

1. C'est évident : il suffit de prendre $N = 0$ et $c = 1$. On a alors $\forall\, n \geqslant 0,\ f(n) \leqslant f(n)$.
2. Si $f = O(g)$ alors il existe $N_1 \in \mathbb{N}$ et $c_1 \in \mathbb{R}_+$ tels que :

   $$\forall\, n \geqslant N_1,\ f(n) \leqslant c_1 \times g(n)$$

   De même si $g = O(h)$ alors il existe $N_2 \in \mathbb{N}$ et $c_2 \in \mathbb{R}_+$ tels que :

   $$\forall\, n \geqslant N_2,\ g(n) \leqslant c_2 \times h(n)$$

   Posons alors $N = \max(N_1, N_2)$. On en déduit que :

   $$\forall\, n \geqslant N,\ f(n) \leqslant c_1 \times g(n) \leqslant c_1 c_2 \times h(n)$$

   Il suffit alors de prendre $c = c_1 c_2$ et on a donc bien $f = O(h)$.
3. Comme $f = O(g + h)$, il existe $N_1 \in \mathbb{N}$ et $c \in \mathbb{R}_+$ tels que :

   $$\forall\, n \geqslant N_1,\ f(n) \leqslant c_1 \times (g(n) + h(n))$$

De plus comme $\lim_{n\to+\infty} \frac{h(n)}{g(n)} = 0$, il existe $N_2 \in \mathbb{N}$ tel que :

$$\forall\, n \geqslant N_2,\ \frac{h(n)}{g(n)} \leqslant 1 \iff h(n) \leqslant g(n)$$

Posons alors $N = \max(N_1, N_2)$. On en déduit que :

$$\forall\, n \geqslant N,\ f(n) \leqslant 2c_1 \times g(n)$$

et il suffit de prendre $c = 2c_1$ pour montrer que $f = O(g)$.

4. Si $f = O(ag)$ avec $a \in \mathbb{R}_+$, il existe $N \in \mathbb{N}$ et $c_1 \in \mathbb{R}_+$ tels que :

$$\forall\, n \geqslant N,\ f(n) \leqslant c_1 \times ag(n) = (ac_1) \times g(n)$$

On prend donc $c = ac_1$ et la propriété est démontrée : $f = O(g)$.

Application : analysons la complexité temporelle des trois exemples précédents :

Exemple 1 : $\mathrm{T}(n) = a \times n + b$ si $n \geqslant 2$. On a donc $\mathrm{T} = O(an + b)$ (règle 1) $= O(an)$ (règle 3) $= O(n)$ (règle 4), donc :

$$\boxed{\mathrm{T}(n) = O(n)}$$

Exemple 2 : $\mathrm{T}(n) \leqslant a \times \lg(n) + b$ pour $n \in \mathbb{N}^*$. On prend $N = 1$ et $c = 1$ et on a donc $\mathrm{T} = O(a\,\lg(n) + b) = O(a\,\lg(n))$ (règle 3) $= O(\lg(n))$ (règle 4), donc :

$$\boxed{\mathrm{T}(n) = O(\lg(n))}$$

Exemple 3 : c'est la même chose que l'exemple 1 et donc

$$\boxed{\mathrm{T}(n) = O(n)}$$

4) Analyse de la complexité temporelle

La notation O est très commode pour classer les algorithmes par complexité temporelle croissante.

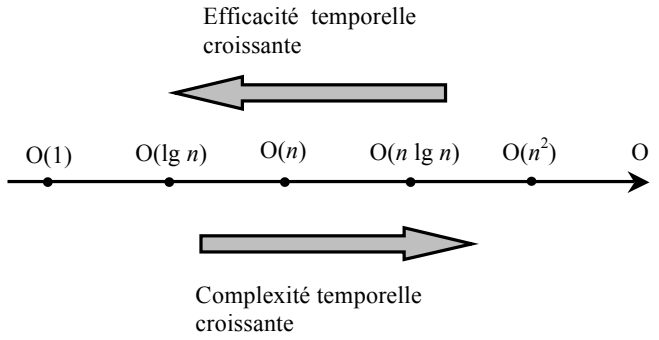

Figure 2 Classement des algorithmes en fonction de leur complexité temporelle.

Remarque :

Lorsque la complexité temporelle est en O(1) cela signifie que, si n est "assez grand", il existe une constante c telle que : $\mathrm{T}(n) \leqslant c$. On dit que l'algorithme s'exécute à **temps constant**.