Notes sur le calcul de la complexité temporelle d'un algorithme

1. Rappel

L'analyse d'un algorithme consiste à évaluer ses besoins en **temps** et en **espace**. Nous nous concentrons ici sur la **complexité temporelle**, c'est-à-dire le nombre d'opérations élémentaires nécessaires en fonction de la taille de l'entrée.

La complexité est toujours exprimée **en fonction de la taille des entrées**. Un algorithme peut dépendre de plusieurs paramètres ($n, m, \ldots$). Dans cette analyse, nous simplifions la présentation en considérant une seule variable, mais dans d'autres contextes il peut être essentiel d'en prendre plusieurs.

Le **temps d'exécution réel** dépend souvent des données fournies. Par exemple, pour une recherche linéaire dans une liste de longueur n, le temps dépend de la position de l'élément cherché. On distingue donc :
- le meilleur cas (élément trouvé immédiatement) ;
- le cas moyen ;
- le pire cas (élément absent de la liste).

Nous analysons en général le **pire cas**, celui où l'algorithme effectue le plus grand nombre d'étapes.

Pour de petites entrées, la différence entre les algorithmes est négligeable. Mais lorsque la taille croît fortement, les écarts deviennent spectaculaires : une recherche linéaire sur 10^9 éléments nécessite 10^9 comparaisons, contre seulement 31 pour une recherche binaire. Nous nous intéressons donc au **comportement asymptotique**, c'est-à-dire la tendance quand $n \to \infty$. On l'exprime à l'aide de la **notation** Θ, qui décrit la croissance dominante d'une fonction.

2. Méthode pour déterminer la complexité

Pour déterminer la complexité asymptotique d'un algorithme, on suit quatre étapes :

1. **Déterminer le coût élémentaire** c_l de chaque ligne l.
2. **Déterminer le nombre d'exécutions** r_l de cette ligne.
3. **Sommer** :
$$T(n) = \sum_{l=1}^{m} c_l \cdot r_l$$
où m est le nombre de lignes et n la taille de l'entrée.
4. **Approximer** $T(n)$ à l'aide de la notation Θ.

2.1 Coût d'une ligne

Le coût dépend du **modèle de calcul** adopté. Sur un ordinateur classique, les opérations suivantes ont un **coût constant** $\Theta(1)$:
- affectation ($\leftarrow$) ;
- opérations arithmétiques (+, -, *, /) ;
- comparaisons (<, <=, =, >=, >) ;
- accès à un élément d'une liste ($a[i]$).

Expression	Coût
$a \leftarrow 0$	$\Theta(1)$
$a \leftarrow a + 2$	$\Theta(1)$
$l[a] + 4$	$\Theta(1)$
$2 \times \text{size}(l)$	$\Theta(1)$ + coût de size(l)

2.2 Nombre d'exécutions d'une ligne

Ce nombre dépend du **contexte**, notamment des boucles et des appels récursifs.

Exemple 1 : boucle simple

algorithme 1
entrée : *liste L de nombres entiers, de taille n, nombre entier M*
sortie : *nombre entier positif ou nul s*

```
a ← 0
b ← 0
Pour e dans L
    a ← e
    b ← e + 1
```

Les deux premières lignes s'exécutent une fois. Les lignes 4–5 s'exécutent n fois : $\Theta(n)$.

Exemple 2 : boucle bornée

Somme des pairs
entrée : *Entier n*
sortie : *Somme s*

```
s ← 0
Pour i allant de 1 à 2n
    Si i pair
        s ← s + i
```

Les lignes 1–3 s'exécutent $2n$ fois, soit $\Theta(n)$. La ligne 4 s'exécute environ n fois.

Exemple 3 : décrémentation

Décrémentation par 3
entrée : *Entier n*
sortie : *Aucune*

```
i ← n
Tant que i > 0
    i ← i − 3
```

La boucle se répète $\lfloor n/3 \rfloor$ fois : $\Theta(n)$.

Exemple 4 : boucles imbriquées

Produit double
entrée : *Entier n*
sortie : *Somme s*

```
s ← 0
Pour i allant de 1 à n
    Pour j allant de i à n
        s ← s + i × j
```

Le corps interne s'exécute

$$\sum_{i=1}^{n}(n-i+1) = \frac{n(n+1)}{2}$$

fois : $\Theta(n^2)$.

Exemple 5 : division entière

Division répétée
entrée : *Entier n*
sortie : *Aucune*

```
i ← n
Tant que i > 0
    i ← i/3
```

La boucle s'exécute environ $\log_3(n)$ fois : $\Theta(\log n)$.

Exemple 6 : récursion linéaire

Algorithme récursif simple `algor`
entrée : *Entier n*
sortie : *Valeur entière*

```
Si n ≤ 0
    Sortir : 0
s ← algor(n − 1)
Sortir : s + 1
```

Chaque appel engendre un autre jusqu'à $n = 0$. Il y a donc $n + 1$ appels : $\Theta(n)$.

2.3 Combinaison des coûts

Une fois les coûts c_l et répétitions r_l déterminés :

$$T(n) = \sum_{l=1}^{m} c_l \cdot r_l$$

Boucles imbriquées
entrée : *Entier n* sortie : *Aucune*

$s \leftarrow 0$
Pour *i allant de* 1 *à n*
  Pour *j allant de i à n*
  $s \leftarrow s + i \times j$

Pour chaque ligne de cet algorithme :

- Ligne 1 : coût $c_1 = 1$, exécutée une fois $\Rightarrow 1 \cdot 1 = 1$.
- Ligne 2 : coût approximatif $c_2 = 3$ opérations (comparaison $i \leq n$, incrément $i \leftarrow i + 1$, affectation) exécutée n fois $\Rightarrow 3n$.
- Ligne 3 : coût $c_3 = 3$, exécutée $\frac{n(n+1)}{2}$ fois $\Rightarrow 3 \cdot \frac{n(n+1)}{2}$.
- Ligne 4 : coût $c_4 = 3$, exécutée $\frac{n(n+1)}{2}$ fois $\Rightarrow 3 \cdot \frac{n(n+1)}{2}$.

En additionnant les contributions :

$$T(n) = 1 + 3n + 6 \cdot \frac{n^2 + n}{2} = 3n^2 + 6n + 1$$

soit une complexité $\Theta(n^2)$.

3. Approximation avec la notation Θ

La notation Θ exprime la croissance asymptotique d'une fonction : on ignore les constantes multiplicatives et additives pour ne garder que le terme dominant.

$$f(n) \in \Theta(g(n)) \iff \exists c_1, c_2 > 0, n_0 : \forall n > n_0,\ c_1 g(n) \leq f(n) \leq c_2 g(n)$$

Autrement dit, $g(n)$ encadre $f(n)$ à la fois par le haut et par le bas, à partir d'une certaine taille d'entrée.

f(n)	$\Theta(g(n))$
3	$\Theta(1)$
$33n + 10$	$\Theta(n)$
$1 + 6n + 3n^2$	$\Theta(n^2)$
$-10n^5 + \frac{n^3}{5} - 10$	$\Theta(n^5)$
$\frac{4}{3} \cdot 2^n + n - n^2$	$\Theta(2^n)$
$2^n + 100$	$\Theta(2^n)$
2^{3n}	$\Theta(8^n)$
$\log_2(n) + n^2$	$\Theta(n^2)$
$10 + \log_2(3n^5)$	$\Theta(\log n)$

4. En résumé

Pour déterminer la complexité d'un algorithme :

1. Identifier les opérations de coût constant ($\Theta(1)$).
2. Compter combien de fois chaque ligne s'exécute ($\Theta(n)$, $\Theta(n^2)$, $\Theta(\log n)$, etc.).
3. Additionner tous les coûts pour obtenir $T(n)$.
4. Simplifier le résultat en ne gardant que le terme dominant et exprimer la croissance en Θ.

L'étape 3 n'est pas toujours nécessaire : pour des algorithmes simples, on peut souvent déduire directement la complexité asymptotique à partir de la structure des boucles ou de la récursion.

Cette méthode fournit une estimation précise et indépendante du matériel, utile pour comparer objectivement les algorithmes lorsque la taille des données devient très grande.