Chapitre I :

Rappel sur la Complexité Algorithmique

L'objectif de ce premier chapitre est d'introduire la notion de complexité algorithmique et de fournir les méthodes et les outils permettant d'analyser la complexité des algorithmes.

1. Définition de la complexité d'un algorithme

La complexité d'un algorithme est une mesure permettant d'évaluer la performance de cet algorithme en fonction de la taille des données du problème à traiter. Nous distinguons:

- **La complexité en temps**, qui mesure le temps nécessaire à l'exécution d'un algorithme.
- **La complexité en espace**, qui mesure la taille mémoire nécessaire à l'exécution d'un algorithme.

L'analyse de la complexité consiste à déterminer ces deux grandeurs, dans le but de comparer des algorithmes résolvant le même problème. En effet, il est fréquent qu'un même problème possède plusieurs solutions décrites par des algorithmes différents. La comparaison entre ces algorithmes passe par l'étude de leur complexité.

Dans ce chapitre, nous traiterons la complexité temporelle des algorithmes (mais les mêmes notions permettent de traiter la complexité spatiale) dont les coûts d'exécution dépendent des machines sur lesquelles ils s'exécutent. Mais nous ferons, dans ce chapitre, abstraction de ce facteur, pour nous concentrer uniquement sur le coût des actions résultant de l'exécution d'un algorithme, en fonction d'une taille n des données traitées.

2. Types de complexité

Notons *Coût(d)* la complexité temporelle d'un algorithme sur la donnée *d*, et D_n l'ensemble des données de taille *n*. On s'intéresse alors aux quantités suivantes:

a) **Complexité dans le meilleur des cas:** c'est la complexité de l'algorithme dans le cas le plus favorable.

$$Min(n) = Min\{coût(d)/\ d \in D_n\}$$

b) Complexité dans le pire des cas : c'est la complexité de l'algorithme dans le cas le plus défavorable.

$$Max(n) = Max\{co\hat{u}t(d)/\ d \in D_n\}$$

c) Complexité en moyenne : pour définir cette complexité, il faut disposer d'une mesure de probabilité sur l'ensemble des données de taille *n*. Cette mesure ne fait souvent qu'approximer l'espace réel des données, car il est très dur de proposer un modèle des données "réelles". La complexité en moyenne est alors donnée par:

$$Moy(n) = \sum_{d \in D_n} Co\hat{u}t\ (d) \times P(d)$$

Avec $P(d)$ la probabilité d'avoir la données *d* en entrée.

Quand on parle de la complexité d'un algorithme sans préciser laquelle, c'est souvent de la complexité temporelle dans le pire des cas qu'on parle. La complexité dans le meilleur des cas n'est pas très utilisée; la complexité en moyenne est d'une certaine façon celle qui révèle le mieux le comportement "réel" de l'algorithme à condition de disposer d'un modèle de la répartition des données et elle est souvent dure à calculer, même de façon approximative. Cependant, elle ne garantit rien sur le pire des cas qui permet de borner le temps d'exécution d'un algorithme.

3. Démarche à suivre pour calculer la complexité d'un algorithme

Pour calculer la complexité d'un algorithme, il faut procéder comme suit:

1) Evaluer la taille des données nécessaires à l'algorithme.
2) Déterminer les opérations élémentaires (affectations, comparaisons (<=, =, >, ….), opérations arithmétiques (+,×,…), opérations logiques (et, ou, non logique), lecture, écriture,…etc.) qui sont à égalité de coûts (temps constant).
3) Faire la somme des coûts de ses opérations élémentaires.

3.1. Evaluation des coûts séquentiels

a) <u>Structures répétitives</u>

La boucle *POUR* permet d'exécuter plusieurs fois un même traitement avec différentes données. Sa syntaxe se présente comme suit :

Pour i allant de j à k **Faire**
| B ;
Fin ;

$$\Leftrightarrow \begin{cases} i \leftarrow j & C1 \\ i \leq k & C2 \\ B & C3 \\ i \leftarrow i+1 & C4 \end{cases}$$

La complexité d'une boucle *POUR* est égale à la somme sur toutes les itérations de la complexité du corps de la boucle.

$\Longrightarrow \quad Coût_{Pour} = C_1 + C_2 + C_3 + C_4$

Ainsi, le coût de la boucle *POUR* est calculé comme suit:

Instruction	**Nombre d'opérations**	**Nombre de fois**	**Coût**
i← j	1	1	C_1= 1
i<= k	1	(k-j)+1+1	C_2= 1× (k-j +2)
B	C_B	(k-j)+1	C_4= C_B × (k-j +1)
i←i+1	2	(k-j)+1	C_3=2× (k-j +1)

$$Coût_{Pour} = C_1 + C_2 + C_3 + C_4$$

$$\Longrightarrow Coût_{Pour} = 1 + (k-j+2) + C_B \times (k-j+1) + 2 \times (k-j+1)$$

$$\Longrightarrow Coût_{Pour} = (k-j+1)(3+C_B) + 2$$

Où C_B représente le nombre d'opérations élémentaires du bloc d'instructions B.

Remarque: le même principe sera adopté pour calculer le coût des autres boucles .

b) Les instructions alternatives:

Dans le cas des instructions alternatives, le coût sera calculé comme suit:

Si (condition (X) **Alors**
| A ;
Sinon
B ;
Fin ;

$$Coût_{si} = C_X + MAX\ \{C_A\ , C_B\}\ Dans\ la\ pire\ des\ cas$$
$$Coût_{si} = C_X + MIN\ \{C_A\ , C_B\}\ Dans\ le\ meilleur\ des\ cas$$

Où $C_X, C_A et\ C_B$ représentent, respectivement, le nombre d'opérations élémentaires de la condition X et des blocs d'instructions A et B.

Exemple: Soit à calculer la complexité de l'algorithme suivant

```
Algorithme Recherche ;
Var        A : Tableau[1..50] d'entiers ;
           V, i, n: entier ;
           Trouve : Booléen ;
Début
    Lire(n) ;
    Pour i=1 à n Faire
        Lire (A[i]) ;
    Fin ;
    Lire(V) ;
    Trouve ← Faux;
    i←1;
    TQ (i<=n) et (Trouve= Faux) Faire
        Si (A[i]=V) Alors
            Trouve ← Vrai;
            Sinon
            i ← i+1;
        Fin ;
    Fin ;
    Si (Trouve=vrai) Alors
        Ecrire (''La valeur existe dans le tableau'') ;
        Sinon
        Ecrire (''La valeur n'existe dans le tableau'') ;
    Fin ;
Fin ;
```

L'algorithme ci-dessus permet de faire la recherche d'une valeur donnée V dans un tableau de taille *n*.

1) **Complexité au meilleur des cas :** le cas le plus favorable pour l'algorithme *Recherche* est le cas où la valeur recherchée se trouve au niveau de la 1ère case du tableau. Dans ce cas, la complexité de cet algorithme est donnée dans le tableau ci-dessous.

Instruction	**Nombre d'opérations**	**Nombre d'exécutions**
Lire (n)	1	1
i←1	1	1
i<=n	1	n+1
Lire (A[i])	1	n
i←i+1	2	n
Lire (V)	1	1
Trouve ← Faux	1	1
i←1	1	1
(i<=n) et (Trouve= Faux)	3	2
(T[i]=V)	1	1
Trouve ← Vrai	1	1
i ← i+1	2	0
Trouve =vrai	1	1
Ecrire ("La valeur existe dans le tableau")	1	1
Ecrire (" La valeur n'existe pas dans le tableau)	1	0

$$\Rightarrow \boldsymbol{coût} = 1 + 1 + (n + 1) + n + 2n + 1 + 1 + 1 + 6 + 1 + 1 + 0 + 1 + 1 + 0$$

$$\Rightarrow \boldsymbol{coût} = 4n + 16$$

2) **Complexité au pire cas :** le cas le plus défavorable pour l'algorithme *Recherche* est le cas où la valeur recherchée n'existe pas dans le tableau. Dans ce cas, la complexité de cet algorithme est donnée dans le tableau ci-dessous.

Instruction	**Nombre d'opérations**	**Nombre de fois**
Lire (n)	1	1
i←1	1	1
i<=n	1	n+1
Lire (A[i])	1	n
i←i+1	2	n
Lire (V)	1	1
Trouve ← Faux	1	1
i←1	1	1
(i<=n) et (Trouve= Faux)	3	n+1
(T[i]=V)	1	n
Trouve ← Vrai	1	0
i ← i+1	2	n
Trouve =vrai	1	1
Ecrire (…existe..)	1	0
Ecrire (…n'existe pas..)	1	1

$$\Rightarrow \boldsymbol{coût} = \mathbf{1} + \mathbf{1} + (\boldsymbol{n} + \mathbf{1}) + \boldsymbol{n} + \mathbf{2}\boldsymbol{n} + \mathbf{1} + \mathbf{1} + \mathbf{1} + \mathbf{3}(\boldsymbol{n} + \mathbf{1}) + \boldsymbol{n} + \mathbf{0} + \mathbf{2}\boldsymbol{n} + \mathbf{1} + \mathbf{0} + \mathbf{1}$$

$$\Rightarrow \boldsymbol{coût} = \mathbf{10}\boldsymbol{n} + \mathbf{11}$$

3.2. Evaluation du coût des sous-programmes récursifs

Dans le cas des sous-programmes récursifs, le dénombrement consiste le plus souvent à résoudre des équations de récurrence.

Exemple : soit à calculer la complexité de la fonction suivante:

```
Fonction    Factorielle (n :entier) : entier ;
Début
    Si (n=0) ou (n=1) Alors
        Factorielle ← 1 ;
        Sinon
        Factorielle ← n × Fact (n-1);
    Fin;
Fin;
```

A chaque appel récursif, sept opérations élémentaires seront exécutées. En d'autres termes, toutes les instructions de la fonction *Factorielle* seront exécutées à l'exception de l'instruction *'Factorielle ← 1'* qui sera exécutée une et une seule fois et ce lorsque le critère d'arrêt de la récursivité sera atteint (Arrêt des appels récursif lorsque n=1). Dans ce cas, quatre opérations élémentaires seront exécutées. Dans ce cas, le coût $\boldsymbol{C(n) = C(1) = 4}$.

Ainsi, lorsque **n>1**, le coût sera calculé comme suit:

$$\boldsymbol{C(n) = 7 + C(n-1)}$$
$$\boldsymbol{C(n) = 7 + \big(7 + C(n-2)\big)}$$
$$\boldsymbol{C(n) = 7 + \big(7 + (7 + C(n-3))\big)}$$
$$\boldsymbol{C(n) = 7 \times 3 + C(n-3)}$$
$$\vdots$$
$$\boldsymbol{C(n) = 7 \times (n-1) + C(n-(n-1))}$$
$$\boldsymbol{C(n) = 7 \times (n-1) + C(1)}$$
$$\boldsymbol{C(n) = 7n - 7 + 4}$$
$$\boldsymbol{C(n) = 7n - 3}$$

4. Expression de la complexité

Quand nous calculons la complexité d'un algorithme, nous ne calculons généralement pas sa complexité exacte mais son ordre de grandeur. Pour ce faire, nous avons besoin de notations asymptotiques. En d'autres termes, nous sommes plus intéressés par un comportement asymptotique (que se passe t'il quand n tend vers l'infini?) que par un calcul exact pour *n* fixé. Par exemple, si un algorithme exécute $(n^2 + 2n + 5)$ opérations, on

retiendra juste que l'ordre de grandeur est n^2. On utilisera donc les notations classiques sur les ordres de grandeur.

Soient f et g deux fonctions de N dans N.

– On dit que f positive est asymptotiquement Majorée (ou dominée) par g et on écrit :

$$f(n) = O\big(g(n)\big)$$

Lorsqu'il existe une constante c, c>0 tel que pour un 'n'assez grand, on a $f(n) \leq c \times g(n)$

En d'autres termes:

$$f(n) = O\big(g(n)\big) <=> \exists\, c > 0\ et\ n_0 > 0\ tel\ que :\ 0 \leq f(n) \leq c \times g(n), \forall\, n \geq n_0$$

– On définit asymptotiquement la minoration de f par g et on écrit $f(n) = \Omega\big(g(n)\big)$ comme suit:

$$f(n) = \Omega\big(g(n)\big) <=> \exists\, c > 0\ et\ n_0 > 0\ tel\ que :\ 0 \leq c \times g(n) \leq f(n), \forall\, n \geq n_0$$

– On dit que f est de même ordre de grandeur que g et on écrit $f(n) = \Theta\big(g(n)\big)$ lorsque :

$$f(n) = O\big(g(n)\big)\ et\ g(n) = O\big(f(n)\big)$$

En d'autres termes:

$$f(n) = \Theta\big(g(n)\big) <=> \exists\, c_1, c_2 > 0\ et\ n_0 > 0\ tel\ que :\ 0 \leq c_1 \times g(n) \leq f(n) \leq c_2 \times g(n), \forall\, n$$

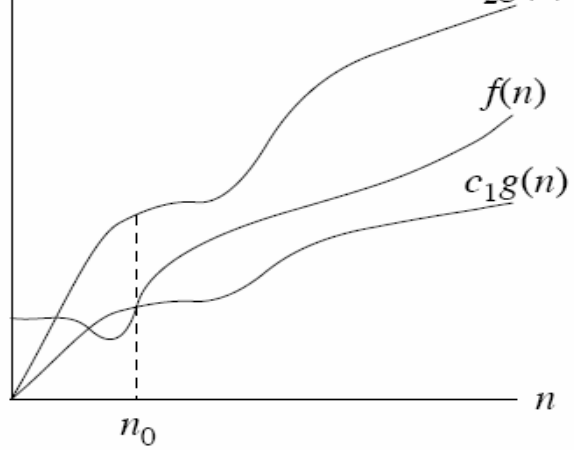

Figure 1.1: Interprétation géométrique des notations asymptotiques

Exemples :

1) **Montrons que $5\,n^2 + 3\,n + 4 = O(n^2)$**

On doit déterminer deux constantes c et n_0 tel que:

$$0 \leq 5\,n^2 + 3\,n + 4 \leq c \times n^2, \qquad pour\ n \geq n_0 \ \ldots\ldots\ldots \quad (1)$$

$$g(n) = n^2 \quad \textbf{\textit{et}} \quad f(n) = 5n^2 + 3\,n + 4$$

Divisons par n^2: $\quad 0 \leq 5 + \frac{3}{n} + \frac{4}{n^2} \leq c$

On peut par exemple choisir n_0=1, d'où l'on déduit que l'inégalité fonctionne pour $c = 12$.

En remplaçant *c* par sa valeur dans (1), nous obtenons ce qui suit:

$$=>\ 0 \leq 5\,n^2 + 3\,n + 4 \leq 12n^2, \qquad pour\ n \geq n_0$$

2) **Montrons que $5\,n^2 + 3\,n + 4 = \Omega(n^2)$**

On doit déterminer deux constantes c et n_0 tel que :

$$0 \leq c \times n^2 \leq 5\,n^2 + 3\,n + 4, \qquad pour\ n \geq n_0 \ \ldots\ldots\ldots \quad (2)$$

$$g(n) = n^2 \quad \textbf{\textit{et}} \quad f(n) = 5n^2 + 3\,n + 4$$

Divisons par n^2:

$$0 \leq c \leq 5 + \frac{3}{n} + \frac{4}{n^2}$$

On peut par exemple choisir n_0=1 et $c = 1$ (on peut aussi choisir $c_1 = 2, 3, 4\ ou\ 5$).

Ainsi, en remplaçant *c* par sa valeur dans (2), nous obtenons ce qui suit:

$$=>\ 0 \leq n^2 \leq 5\,n^2 + 3\,n + 4, \qquad pour\ n \geq n_0$$

3) **Montrons que $5\,n^2 + 3\,n + 4 = \Theta(n^2)$**

On doit déterminer trois constantes c_1, c_2 et n_0 avec :

$$c_1 \times n^2 \leq 5\,n^2 + 3\,n + 4 \leq c_2 \times n^2, \qquad pour\ n \geq n_0 \ \ldots\ldots\ldots \quad (3)$$

$$g(n) = n^2 \quad \textbf{\textit{et}} \quad f(n) = 5n^2 + 3\,n + 4$$

Divisons par n^2:

$$c_1 \leq 5 + \frac{3}{n} + \frac{4}{n^2} \leq c_2$$

On peut par exemple choisir $c_1 = 5$ (on peut aussi choisir $c_1 = 1$) et n_0=1, d'où l'on déduit que l'inégalité fonctionne pour $c_2 = 12$.

En remplaçant c_1 et c_2 par leurs valeurs dans (3), nous obtenons ce qui suit:

$$n^2 \leq 5\,n^2 + 3\,n + 4 \leq 12n^2, \qquad pour\ n \geq n_0$$

5. Principales classes de complexité

Les algorithmes usuels peuvent être classés en un certain nombre de grandes classes de complexité:

- Les algorithmes ***sub-linéaires***, dont la complexité est en général en ***O(log n).*** C'est le cas de la recherche d'un élément dans un ensemble ordonné fini de cardinal *n*.

- Les algorithmes ***linéaires*** en complexité ***O(n)*** ou en ***O(n log n)*** qui sont considérés comme rapides, comme l'évaluation de la valeur d'une expression composée de ***n*** symboles ou les algorithmes optimaux de tri.

- Plus lents sont les algorithmes de complexité située entre ***O(n^2)*** et ***O(n^3)***, c'est le cas de la multiplication des matrices et du parcours dans les graphes.

- Au delà, les algorithmes ***polynomiaux*** en ***O(n^k)*** pour k > 3 sont considérés comme lents, sans parler des algorithmes exponentiels ***O(k^n)*** (dont la complexité est supérieure à tout polynôme en *n*) que l'on s'accorde à dire impraticables dès que la taille des données est supérieure à quelques dizaines d'unités.

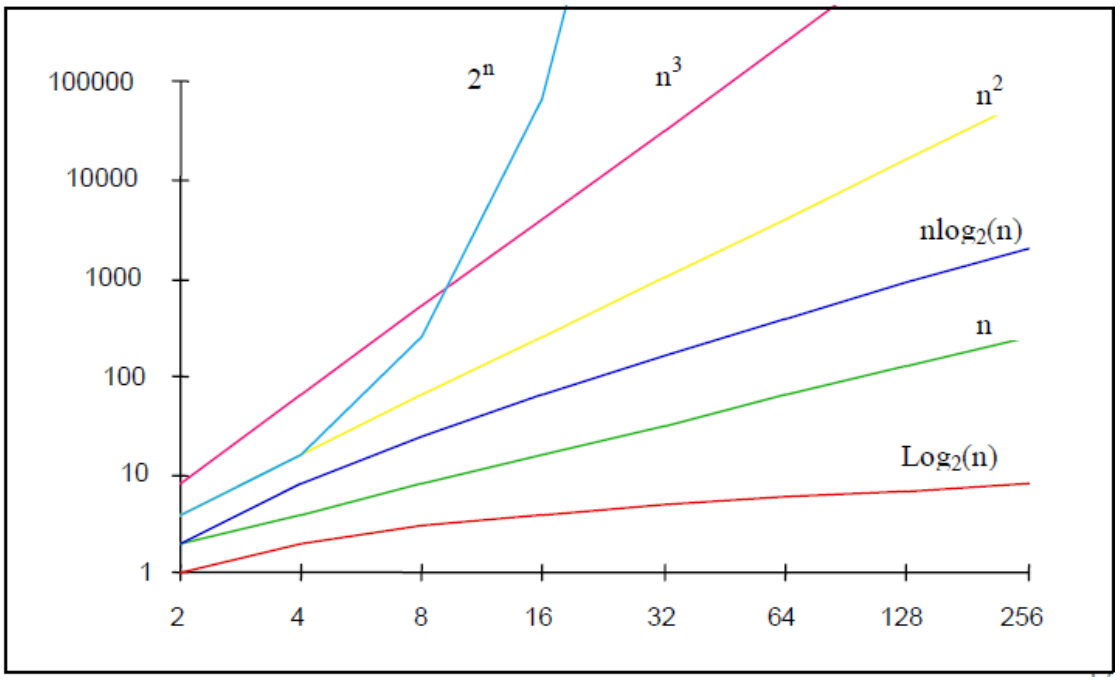

Figure 1.2: Efficacité comparée des principales classes de complexité

6. Conclusion

La complexité algorithmique permet de s'informer sur l'efficacité intrinsèque d'un algorithme, c'est-à-dire sur la performance d'un algorithme en dehors de l'environnement dans lequel sera implémenté ce dernier : machine physique, système d'exploitation, compilateurs de programmes, …etc. Il est donc toujours utile de rechercher des algorithmes efficaces, même si les progrès technologiques accroissent les performances du matériel.