Chapitre 2

Complexité: Concepts & Outils

I. Introduction:

L'efficacité d'un algorithme se mesure par *le temps d'exécution en fonction de la taille des données d'entrée et la place mémoire nécessaire à son exécution.*

La détermination de ces deux fonctions s'appelle l'analyse de la complexité.

Dans la suite nous n'étudierons que le temps d'exécution. Pour le mesurer comme une fonction de la taille des données il faut se donner une unité de mesure.
Pour simplifier le modèle on considère comme temps constant les opérations élémentaires exécutés par une machine (opérations arithmétiques, logiques, affectations, etc.).

L'analyse consiste alors à exprimer le nombre d'opérations effectuées comme une fonction de la taille des données d'entrée.

Exemple

Début
  K ← 1
  P ← 0
    Pour i de 0 à N faire
    P ← P + K * a_i
    K ← K * x
    FinPour
Fin

Coût de l'algorithme:

- (n+1) additions
- 2 (n+1) multiplications

II. Rappels:

II. 1 Algorithme & Algorithmique:

Un algorithme est une suite ordonnée d'opérations ou d'instruction écrites pour la résolution d'un problème donné.

Un algorithme est une suite d'actions que devra effectuer un automate pour arriver à partir d'un état initial, en un temps fini, à un résultat.

L'algorithmique désigne le processus de recherche d'algorithme.

II. 2 Structures de Données:

Une structure de données indique la manière d'organiser des données dans la mémoire. Le choix d'une structure adéquate dépend généralement du problème à résoudre.

Il existe deux types de structures de données :

— Statiques : l'espace mémoire est alloué dès le début de résolution du problème. Exemple : les tableaux

— Dynamiques : l'espace mémoire est alloué au fur et à mesure des besoins de la résolution du problème. Exemple : liste chainée, pile, file, . . .

La notion des pointeurs est nécessaire pour la gestion des liens entre les données d'un problème en mémoire.

II. 3 Propriétés d'un algorithme:

De nombreux outils formels ou théoriques ont été développés pour décrire les algorithmes, les étudier, exprimer leurs qualités, pouvoir les comparer :

Ainsi, pour décrire les algorithmes, des structures algorithmiques ont été mises en évidence : structures de contrôle (boucle, conditionnelle, ...) et structures de données (variables, listes, ...).

Pour justifier de la qualité des algorithmes, les notions de correction, de complétude et de terminaison ont été mises en place.

Enfin, pour comparer les algorithmes entre eux, une théorie de la complexité des algorithmes a été définie.

Terminaison : C'est l'assurance que l'algorithme terminera en un temps fini.

Correction : Étant donnée la garantie qu'un algorithme terminera, la preuve de correction doit apporter l'assurance que si l'algorithme termine en donnant une proposition de solution, alors cette solution est correcte (solution au problème posé).

Complétude : La preuve de complétude garantit que, pour un espace de problèmes donné, l'algorithme, s'il termine, donnera des propositions de solutions correctes

Efficacité: Il faut que le programme exécute sa tâche avec efficacité c'est à dire avec un coût minimal. Le coût pour un ordinateur se mesure en termes de temps de calcul et d'espace mémoire nécessaire.

III.A propos du temps d'exécution:

III. 1 Motivation:

la complexité est en quelque sorte la maîtrise de l'algorithmique. Elle vise à déterminer, sous un certain nombre de conditions, quel sont les temps de calcul et espace mémoire requis pour obtenir un résultat donné.
Une bonne maîtrise de la complexité se traduit par des applications qui tournent en un temps prévisible et sur un espace mémoire contrôlé.

A l'inverse, une mauvaise compréhension de la complexité débouchent sur des latences importantes dans les temps de calculs, ou encore des débordements mémoire conséquents, qui au mieux, « gèlent » les ordinateurs (« swap » sur le disque dur) et au pire font carrément « planter »la machine.

III. 2 Analyse du temps d'exécution:

Le temps d'exécution d'un programme est la somme principalement de deux composantes : le temps de calcul et le temps d'accès aux données. Il dépend ainsi de plusieurs facteurs :

1. du type d'ordinateur utilisé ;
2. du langage utilisé (représentation des données) ;
3. la hiérarchie mémoire et le profile d'accès aux données ;
4. la complexité abstraite de l'algorithme sous-jacent.

Pour simplifier le modèle, on va s'abstraire des trois premiers facteurs.
Pour ce faire, il faut être en mesure de donner un cadre théorique à l'algorithmique.

Pour le premier point ceci se fait par la définition d'un ordinateur « universel » (une machine de Turing) qui bien qu'extrêmement simple peut reproduire le comportement de n'importe quel ordinateur existant.

Pour s'abstraire du second point, on regardera des classes d'équivalence de complexité plutôt que la complexité elle-même, ce qui permettra de ne pas se préoccuper des constantes qui interviennent dans les changements de représentations et dans la définition des opérations élémentaires.

Pour le troisième point, on va s'en passer aussi par machine universelle de Turing qui suppose que la mémoire (« ruban ») est plate et le coût d'accès est uniforme.

La complexité abstraite se mesure en fonction de la taille des entrées (la dimensions caractéristiques des données)

— Pour une liste, le paramètre est le cardinal de la liste.

— Pour une matrice carrée de taille n c'est n le paramètre.

— Pour un polynôme, c'est son degré.

— pour un arbre, on peut considérer son hauteur, ou le nombre de nœuds ;

— Enfin dans certains cas, il n'est pas possible de ne faire intervenir qu'une donnée. Par exemple pour un graphe, le nombre de sommets et d'arêtes peuvent être tous les deux des facteurs importants et pourtant complètement indépendants.

IV. Notions Mathématiques:

IV. 1 Complexité asymptotique:

Soient deux algorithmes A et B résolvant le même problème de complexité respectives 100n et n^2.
Quel est le plus efficace ?

— Pour $n < 100$, B est plus efficace.
— Pour $n = 100$, A et B ont la même efficacité.
— Pour $n > 100$, A est plus efficace.

Notons que plus n devient grand, plus A est efficace devant B. Si les tailles sont « petites », la plupart des algorithmes résolvant le même problème se valent. C'est le comportement de la complexité d'un algorithme quand la taille des données devient grande qui est important.

On appelle ce comportement la complexité asymptotique.

IV. 2 Estimation asymptotique:

La complexité d'un algorithme est une estimation (ou approximation) en limite supérieure du temps nécessaire à l'exécution. On se contente donc de donner des ordres de grandeur.

✓ On ignore les facteurs constants : un calcul qui prend 10 n étapes sur une machine A, peut ne prendre que 5n sur une machine B.

✓ On ignore aussi les étapes d'initialisation. Si un calcul prend $2n^2 + 5n$, on ignore le temps 5n, car il est négligeable devant $2n^2$ pour de grandes valeurs de n.

IV. 3 Notations de « Landau »:

Pour calculer la complexité d'un algorithme, nous ne calculerons généralement pas sa complexité exacte, mais son ordre de grandeur. Pour ce faire, nous avons besoin des notations asymptotiques.

Soit un problème $\mathcal{P}$ de taille n. On note $\mathcal{P}$(n) une instance de $\mathcal{P}$.

On veut calculer la complexité temporelle T (n) de l'algorithme A qui résout $\mathcal{P}$.

Les notations asymptotiques sont définies comme suit :

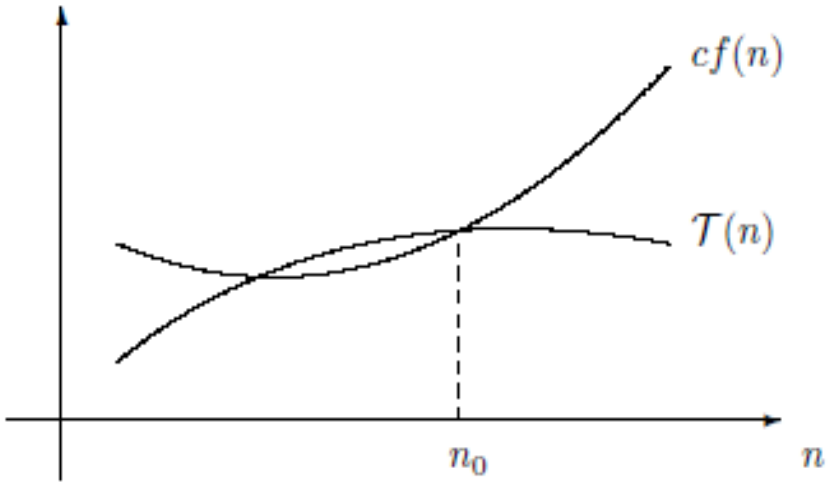
IV. 3. 1 Limite supérieur : Grand-$\mathcal{O}$

La notation $\mathcal{O}$ sert à estimer le temps d'exécution d'un algorithme. Elle permet de trouver un majorant à $\mathcal{T}(n)$.

On dit que $\mathcal{T}(n)$ est en $\mathcal{O}(f(n))$ si et seulement s'il existe un entier n_0 et une constante $c > 0$ tel que

$$\forall n \geq n_0, \text{ on a : } \mathcal{T}(n) \leq cf(n)$$

$\mathcal{O}(f(n))$ est l'ensemble des fonctions $\mathcal{T}(n)$ qui peuvent être bornées supérieurement par **c f(n)** pour **n** suffisamment grand.



Exemple:

$T(n) = 60n^2 + 5n + 1 \in \mathcal{O}(n^2)$.

car on peut trouver $c = 66$ et $n_0 = 1$ / $\forall n \geq 1$ on a $T(n) \leq 66n^2$.

La notation grand-$\mathcal{O}$ peut être obtenue à partir de l'expression $T(n)$ en ne conservant que le terme le plus fort de l'expression est ignorer les autres.

Exemples:

$T(n) = 3n^2 + 10n + 1 \Rightarrow T(n) = \mathcal{O}(n^2)$

$T(n) = n^3 + 10000n \Rightarrow T(n) = \mathcal{O}(n^3)$

$T(n) = 25 \Rightarrow T(n) = \mathcal{O}(1)$: le nombre d'opérations exécutées ne dépend pas de la taille du problème.

$T(n) = 3\log(n) + 10 \Rightarrow T(n) = O(\log(n))$

On suppose qu'on dispose d'un algorithme dont le temps d'exécution est décrit par la fonction $T(n) = 3n^2 + 10n + 10$. L'utilisation des règles de la notation $\mathcal{O}$ nous permet de simplifier en :

$$\mathcal{O}(T(n)) = \mathcal{O}(3n^2 + 10n + 10) = \mathcal{O}(3n^2) = \mathcal{O}(n^2)$$

Pour $n = 10$ nous avons :

— Temps d'exécution de

$$3n2 : \frac{3(10)^2}{3(10)^2 + 10(10) + 10} = 73,2\%$$

— Temps d'exécution de

$$10n : \frac{10(10)}{3(10)^2 + 10(10) + 10} = 24,4\%$$

— Temps d'exécution de

$$10 : \frac{10}{3(10)^2 + 10(10) + 10} = 2,4\%$$

Le poids de $3n^2$ devient encore plus grand quand $n = 100$, soit $96,7\% \Rightarrow$ on peut négliger les quantités $10n$ et 10. Ceci explique les règles de la notation $\mathcal{O}$.

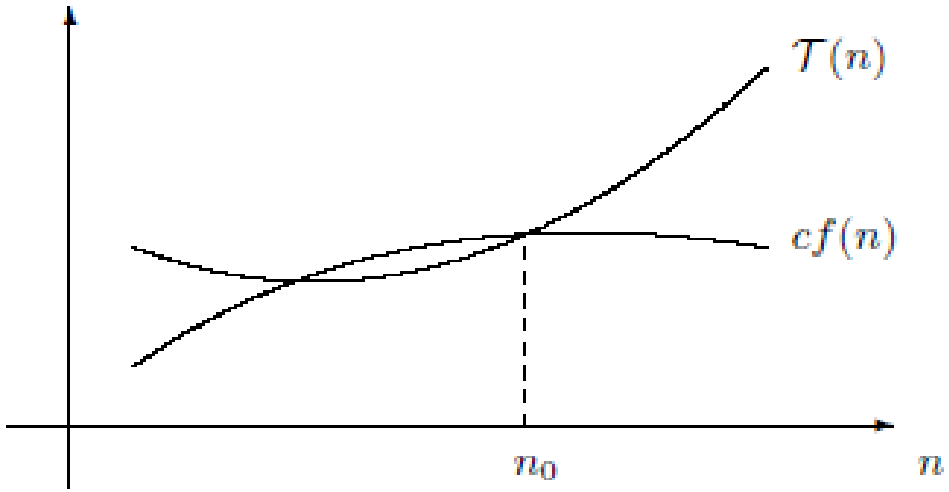
IV. 3. 2 Limite inférieur : Grand-Ω

La notation « Omega » permet de trouver un minorant.

On a T (n) $\in$ Ω(f(n)), si et seulement s'il ∃ deux constantes n_0 et c > 0 telles que :

$$\forall n \geq n_0, \text{ on a } T(n) \geq cf(n)$$

Ω(f(n)) est l'ensemble de toutes les fonctions T (n) bornées inférieurement par cf(n) pour n suffisamment grand.



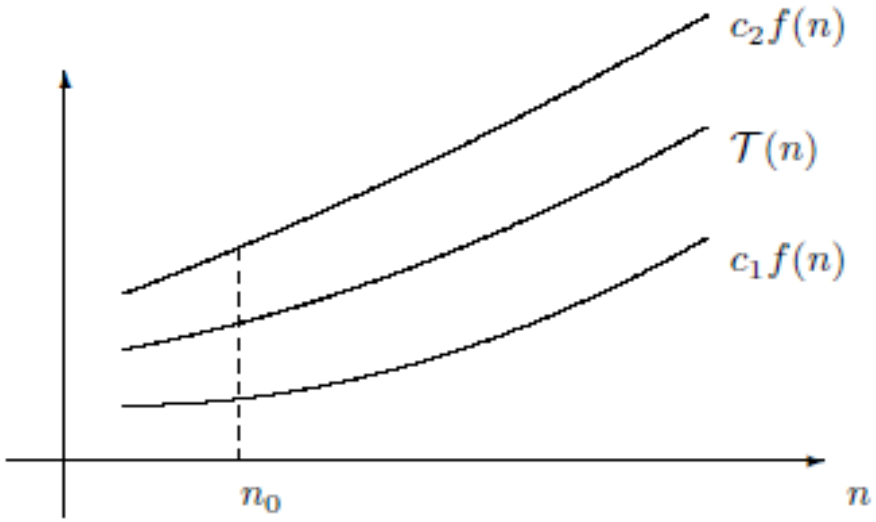
IV. 3. 3 Ordre exact: Grand-Θ

Si une fonction peut être majorée et minorée par une même fonction en notation asymptotique, alors on parle d'ordre exacte (notation « Théta »).

Une fonction $T(n) \in \Theta(f(n))$ si et seulement si elle vérifie à la fois $T(n) \in \mathcal{O}(f(n))$ et $T(n) \in \Omega(f(n))$.

C'est-à-dire il $\exists$ deux constantes c_1 et c_2 telles que :

$\forall\ n \geq n_0$, on a $c_1 f(n) \leq T(n) \leq c_2 f(n)$



IV. 3. 4 Utilisation de la limite pour déterminer l'ordre:

$$\lim_{n\to+\infty}\frac{g(n)}{f(n)}=\begin{cases}0 & \text{alors } g(n)\in\mathcal{O}(f(n))\\ c>0 & \text{alors } g(n)\in\Theta(f(n))\\ +\infty & \text{alors } g(n)\in\Omega(f(n))\end{cases}$$

Exemple 1

Soient $f(n)=n\ log(n)+log(n)$ et $g(n)=\sqrt{n}\ log(n)$

$$\Rightarrow f(n)=\Omega(g(n)) \text{ car } \lim_{n\to+\infty}\frac{f(n)}{g(n)}\to+\infty$$

Exemple 2

Soient $f(n) = \frac{n^3}{2}$ et $g(n) = 37n^2 + 120n + 17$

Monter que $g(n) = \mathcal{O}(f(n))$ et $f(n)$ n'est pas $\mathcal{O}(g(n))$.

$$\lim_{n \to +\infty} \frac{g(n)}{f(n)} = \lim_{n \to +\infty} \frac{37\ n^2 + 120\ n + 17}{n^3/\ 2}$$

$$= \lim_{n \to +\infty} 37/n + 120/n^2 + 17/n$$

0 0 0

$$= 0$$

$$\Rightarrow \quad g(n) = \mathcal{O}(f(n))$$

IV. 3. 5 Propriétés:

- Reflèxivité

$$f(n) = \mathcal{O}(f(n))$$

- Transitivité

$$f(n) = \mathcal{O}(g(n)) \text{ et } g(n) = \mathcal{O}(h(n)) \text{ alors } f(n) = \mathcal{O}(h(n))$$

de même pour Ω et Θ

- Produit par un scalaire

$$a > 0, \qquad a\,\mathcal{O}(f(n) = \mathcal{O}(f(n))$$

— **Somme**

$$\mathcal{O}(f(n)+g(n)) = \mathcal{O}(\max_{n\to\infty}(f(n),g(n)))$$

exemple : $\mathcal{O}(n^2+n) = \mathcal{O}(n^2)$ et $\mathcal{O}(n\ log(n)+n) = \mathcal{O}(n\ log(n))$

— **Produit**

$$\mathcal{O}(f(n)) \times \mathcal{O}(g(n)) = \mathcal{O}(f(n).g(n))$$

— **Symétrie** Elle n'est vraie qu'avec Θ :

$$f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$$

Par contre,

$$f(n) = \Omega(g(n) \Leftrightarrow g(n) = \mathcal{O}(f(n))$$

IV. 4 Complexité en temps d'un algorithme:

La complexité d'un algorithme est la mesure du nombre d'opérations fondamentales qu'il effectue sur un jeu de données.

La complexité est exprimée comme une fonction de la taille du jeu de données.

La complexité d'un algorithme est souvent déterminée à travers une description mathématique du comportement de cet algorithme.

Soit D_n l'ensemble des données de taille n et soit C(d) le coût d'exécution de l'algorithme sur la donnée d de taille n.

On définit plusieurs mesures pour caractériser le comportement d'un algorithme.

- Complexité au meilleur :

La complexité au meilleur, ou complexité dans le meilleur cas, est le plus petit nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille fixée, ici à n :

$$\mathcal{T}_{min}(n) = \min_{d \in Dn} C(d)$$

- Complexité au pire:

La complexité au pire, ou complexité dans le pire cas, dite aussi cas le plus défavorable (worst-case en anglais), est le plus grand nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille fixée, ici à n :

$$\mathcal{T}_{max}(n) = \max_{d \in Dn} C(d)$$

- Complexité en moyenne:

La complexité en moyenne est la moyenne des complexités de l'algorithme sur des jeux de données de taille n :

$$T_{moy}(n) = \Sigma \Pr(d)\ C(d)\ ;\ d \in D_n$$

où Pr(d) est la probabilité d'avoir la donnée d en entrée de l'algorithme.

- Synthèse:

C'est l'analyse pessimiste ou au pire qui est généralement adoptée.

En effet, de nombreux algorithmes fonctionnent la plupart du temps dans la situation la plus mauvaise pour eux.

L'analyse au pire des cas donne une limite supérieure de la performance et elle garantit qu'un algorithme ne fera jamais moins bien que ce qu'on a établi.

Un algorithme est dit optimal si sa complexité est la complexité minimale parmi les algorithmes de sa classe.

Même si on s'intéresse quasi-exclusivement à la complexité en temps des algorithmes, l est parfois intéressant de s'intéresser à d'autres ressources, comme la complexité en espace (taille de l'espace mémoire utilisé), la largeur de bande passante requise, . . . etc.

- Classes de complexité:

Les algorithmes usuels peuvent être classés en un certain nombre de grandes classes de complexité. Les complexités les plus utilisées sont :

Classe	Complexité	Exemple
Constante	$\mathcal{O}(1)$	Accéder au premier élément d'un ensemble de données
Logarithmique	$\mathcal{O}(\log n)$	Couper un ensemble de données en deux parties égales, puis couper ces moitiés en deux parties égales, etc.
Linéaire	$\mathcal{O}(n)$	Parcourir un ensemble de données
Quasi-linéaire	$\mathcal{O}(n \log n)$	Couper répétitivement un ensemble de données en deux et combiner les solutions partielles pour calculer la solution générale

Quadra-tique	$\mathcal{O}(n^2)$	Parcourir un ensemble de données en utilisant deux boucles imbriquées	⇑	
Cubique	$\mathcal{O}(n^3)$	Multiplier deux matrices carrées	‖	
…	…	…	‖ ‖	Complexité
Polyno-miale	$\mathcal{O}(n^p)$	Parcourir un ensemble de données en utilisant P boucles imbriquées	‖ ‖	polynômiale
…	…	…	⇓	
Exponen-tielle en 2^n	$\mathcal{O}(2^n)$	Générer tous les sous-ensembles possibles d'un ensemble de données	⇑ ‖ ‖	Complexité exponen-tielle
…	…	…		
Exponen-tielle en n^n	$\mathcal{O}(n^n)$	…	‖ ⇓	

IV. 5 Calcul de la complexité:

- Cas d'un traitement conditionnel:

```
si <condition> alors
                <traitement 1>
           sinon
                <traitement 2>
finsi
```

La complexité de cette séquence vaut

$$\mathcal{O}(T_{<condition>}) + \max(\mathcal{O}(T_{<traitement\ 1>}), \mathcal{O}(T_{<traitement\ 2>}))$$

- Cas d'un traitement itératif:

```
tant que <condition> faire
        <traitement>
fintantque
```

La complexité de cette boucle vaut :

$$Nbr\_itrations \times (\mathcal{O}(T_{<condition>}) + \mathcal{O}(T_{<traitement>})$$

pour i de $<ind\_Deb>$ $<ind\_Fin>$ faire

$<traitement>$

finpour

La complexité de cette boucle vaut :

$$T = \sum_{i=ind\_Deb}^{ind\_Fin} \mathcal{O}(T_{<traitement>})$$