Introduction

1.1 Complexité

- **Définition d'un algorithme :** est une séquence d'étapes de calcul qui transforment l'entrée en sortie."

- **Définition de la complexité d'un algorithme :** est le nombre d'opérations élémentaires qu'il doit effectuer pour mener à bien un calcul en fonction de la taille des données d'entrée.

- **Définition de l'efficacité d'un algorithme :** l'efficacité d'un algorithme est mesurée par l'augmentation du temps de calcul en fonction de la taille des données .

La taille des données

- La taille des données (ou des entrées) va dépendre du codage de ces entrées.

- Par exemple, en fonction du problème, les entrées et leur taille peuvent être :
- Des éléments: nombre d'éléments
- Des nombres : nombre de bits nécessaires à la représentation de ceux-là ;
- Des polynômes : le degré, le nombre de coefficients non nuls ;
- Des matrices m × n : max(m,n), m.n, m + n ;
- Des graphes : nombre de sommets, nombre d'arcs, produit des deux ;
- Des listes, tableaux, fichiers : nombre de cases, d'éléments ;
- Des mots : leur longueur.

Le temps de calcul

Le temps de calcul d'un programme dépend de plusieurs éléments :

- La quantité de données.
- Leur encodage ;
- De la qualité du code engendré par le compilateur ;
- De la nature et la rapidité des instructions du langage ;
- De la qualité de la programmation ;
- De l'efficacité de l'algorithme.

- la complexité des algorithmes étudiée ne dépendra ni de l'ordinateur, ni du langage utilisé, ni du programmeur, ni de l'implémentation tout simplement, elle dépend de nombre d'opérations fondamentales.
- L'opérations fondamentale dépend de problème à résoudre.

Problème	Opération fondamentale
Recherche d'un élément dans une liste	Comparaison
Tri d'une liste, d'un fichier, …	Comparaisons, déplacements
Multiplication des matrices réelles	Multiplications et additions
Addition des entiers binaires	Opération binaire

Tableau 1 : opérations fondamentales en fonction des problèmes

1.2 Coût des opérations

Pour la complexité en temps, il existe plusieurs possibilités :

- Première possibilité : calculer (en fonction de *n*) le nombre d'opérations élémentaires (addition, comparaison, affectation, …) requises par l'exécution puis le multiplier par le temps moyen de chacune d'elle ;
- Pour un algorithme avec essentiellement des calculs numériques, compter les opérations coûteuses (multiplications, racine, exponentielle, …).
- Sinon compter le nombre d'appels à l'opération la plus fréquente.

1.2.1 Coût en séquentiel

Séquence : $T_{séquence}(n)=\sum T_{éléments\ de\ la\ séquence}(n)$

Alternative : *si C alors J sinon K* $T(n)=T_C(n)+max\{T_J(n),T_K(n)\}$

Itération bornée : *pour i de j à k faire B* $T(n)=(k-j+1).(T_{entête}(n)+T_B(n))+T_{entête}(n)$. Dans l'en-tête est mis l'affectation de l'indice et le test de continuation.

Itération non bornée : *tant que C faire B* $T(n)=Nb_{boucles}.(T_B(n)+T_C(n))+T_C(n)$ avec $Nb_{boucles}$ le nombre de boucles qui s'évalue par méthode inductive.

Répéter B jusqu'à C $T(n)=Nb_{boucles}.(T_B(n)+T_C(n))$

1.2.2 Coût en récursif

La méthode récursive utilise trois étapes à chaque niveau de la récursivité :

- Diviser : le problème en un certain nombre de sous-problèmes ;
- Régner : sur les sous-problèmes en les résolvant de manière récursive. Si la taille du sous-problème est assez petite, il est résolu immédiatement ;
- Combiner : les solutions des sous-problèmes pour produire la solution du problème originel.

1.3 Les différentes mesures de complexité

- Soit A un algorithme, *n* un entier, D_n l'ensemble des entrées de taille *n* , une entrée d $\in D_n$ et $coût_A(d)$ le nombre d'opérations fondamentales effectuées par A avec l'entrée d .
- La complexité dans le meilleur des cas : $Min_A(n) = min\{coût_A(d) / d \in D_n\}$.
- La complexité en pire des cas: $Max_A(n) = max\{coût_A(d) / d \in D_n\}$
- La complexité en moyenne des cas :

$Moy_A(n) = \sum_{d \in D_n} p(d).coût_A(d)$ avec $p(d)$ une loi de probabilité sur les entrées.

Il reste à définir $p(d)$. Une hypothèse est que toutes les entrées ayant une taille donnée sont équiprobables.

D'où : $Moyenne\ uniforme_A(n) = \frac{1}{card(D_n)} \sum_{d \in D_n} coût_A(d)$

1.4 Comparaison entre algorithmes

- pour comparer deux algorithmes, nous allons comparer leur taux de croissance ou ordre de grandeur c.à.d le temps d'exécution de l'algorithme pour un nombre important de données (entrées).

- Un algorithme est plus efficace qu'un autre si son temps d'exécution du cas le plus défavorable (le pire des cas) à un ordre de grandeur inférieur.

- pour la mesure de la complexité la notation O (Notation de Landau ou grand O) est souvent utilisée,

1.4.1Propriétés de grand O

- Si f(n) ∈ O(g(n)), alors kf(n) ∈ O(g(n)) En particulier, $a^b \in O(1)$ et $a^{n+b} \in O(a^n)$
- Si e(n) ∈ O(g(n)) et f(n) ∈ O(h(n)) et si g(n) ∈ O(h(n)) alors e(n)+ f(n) ∈ O(h(n))
- En particulier, $f(n) = \sum_{d=0}^{D} a_d n^d \in O(n^D)$
- Si e(n) ∈ O(g(n)) et f(n) ∈ O(h(n)), alors e(n)f(n) ∈ O(g(n)h(n))

D'autres écritures existent

- Borne inferieur (Ω) : f(n) ∈ Ω(g(n)) si g(n) ∈ O(f(n))
- Borne supérieur et inferieur (Θ): f(n) ∈ Θ(g(n)) si f(n)∈ O(g(n))∩Ω(g(n))

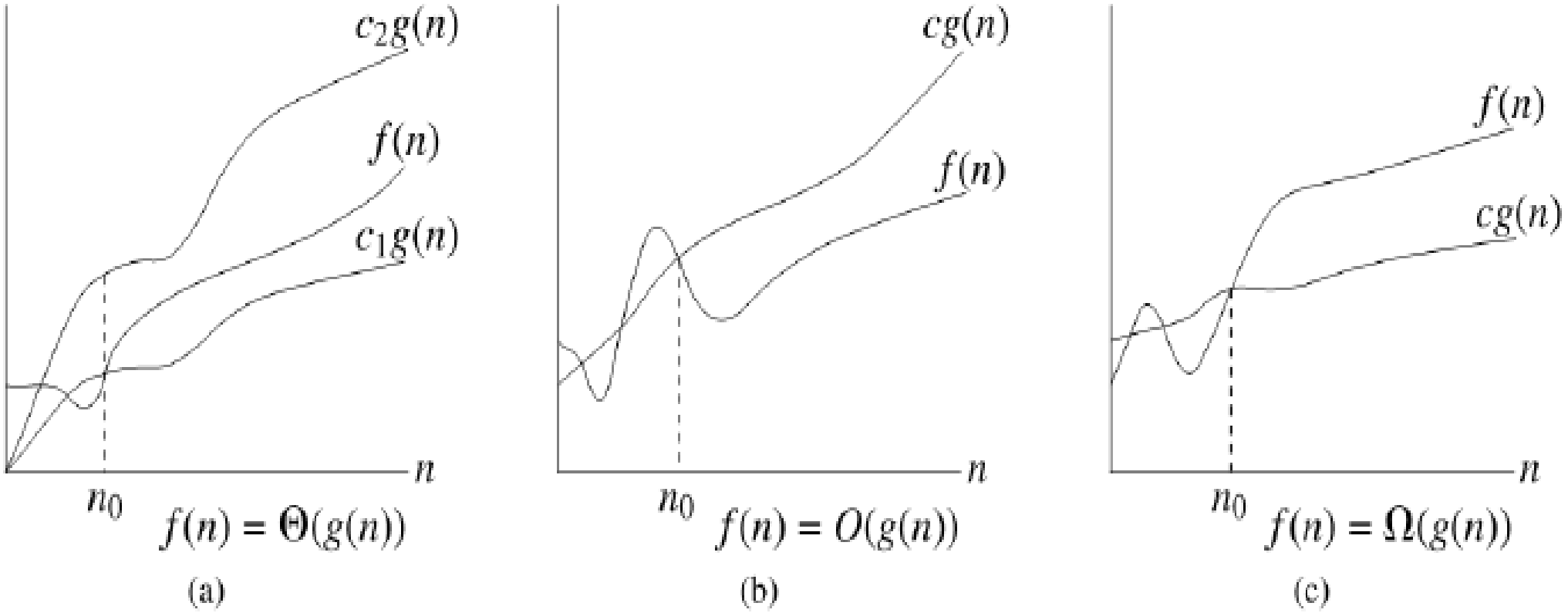

Illustration 1 : Définition graphique des différentes notations utilisées dans le calcul de la complexité

1.5 Classification des algorithmes

Complexité	Description
Complexité **O(1)** Complexité à temps constant	L'exécution ne dépend pas du nombre d'éléments en entrée mais s'effectue toujours en un nombre constant d'opérations
Complexité **O(log(n))** Complexité logarithmique	La durée d'exécution croît légèrement avec n . Ce cas de figure se rencontre quand la taille du problème est divisée par une entité constante à chaque itération.
Complexité **O(n)** Complexité linéaire	C'est typiquement le cas d'un programme avec une boucle de 1 à n et le corps de la boucle effectue un travail de durée constante et indépendante de n .
Complexité **O(n log(n))** Complexité n-logarithmique	Se rencontre dans les algorithmes où à chaque itération la taille du problème est divisée par une constante avec à chaque fois un parcours linéaire des données (ex: l'algorithme de tri "quick sort")
Complexité **O(n^2)** Complexité quadratique	Typiquement c'est le cas d'algorithmes avec deux boucles imbriquées chacune allant de 1 à n et avec le corps de la boucle interne qui est constant.
Complexité **O(n^3)** Complexité cubique	Un algorithmes trois boucles imbriquées.
Complexité **O(n^p)** Complexité polynomiale	Toutes les complexités précédentes sont incluses dans celle-ci. Un algorithme est dit praticable, efficace s'il est polynomial.
Complexité **O(2^n)** Complexité exponentielle	Les algorithmes de ce genre sont dits "naïfs" car ils sont inefficaces et inutilisables dès que n dépasse 50

1.5 Complexité d'un problème

- La complexité d'un problème A est la complexité du meilleur algorithme qui résout A.

Complexité	Exemple
$O(1)$	Accès à un élément de tableau
$O(\log(n))$	Recherche dichotomique
$O(n)$	Recherche dans un tableau non trié
$O(n\log(n))$	Tri rapide
$O(n^2)$	Tri à bulles
$O(n^3)$	Multiplication de matrices
$O(2^n)$	Algorithme du "voyageur de commerce"