CHAPITRE 1: COMPLEXITÉ DES ALGORITHMES ET MESURE DE PERFORMANCE

COURS : COMPLEXITÉ ET OPTIMISATION

RÉALISÉ Par : DR. S. SLATNIA

UNIVERSITÉ DE BISKRA
MASTER D'INFORMATIQUE
2021-2022

PARTIE 1. THÉORIE DE LA COMPLEXITÉ

OBJECTIVE

- Présenter les grands principes de la complexité algorithmique, les qualités demander à un algorithme :
  - Les outils permettant de répondre aux questions suivantes :
    - Le programme s’arrête-t-il de l’exécution (**terminaison**) ?

      il faut montrer que chaque bloc élémentaire décrit ci-dessus termine.
    - Est-ce qu’il réalise effectivement la tâche qu’on lui a demandé (**correct**)?
    - Il faut qu'il soit bien écrit, compréhensible en vue d’une maintenance ou d’une amélioration
- Présenter le critère d’évaluation **d’efficacité** entre les algorithmes, L’algorithme doit être performant :
  - **Le temps de calcul** nécessaire, directement lié au nombre d’opérations élémentaires qu’effectué l’algorithme.
  - **La quantité d’occupation mémoire** nécessaire.

Problème

Donnée → *l'algorithme fournir* (nombre fini d'étapes) → réponse
programme doit finir par s'arrêter

- **Formellement**

Un problème abstrait = l'ensemble < E, A, S > où
E est une entrée (un des cas du problème),
A l'algorithme,
S la sortie (résultat).
Un algorithme = < C entraînent → I >,
C : clauses (règles)
I : instructions (traitements).
Ces règles forment une partition de l'univers**:**
- **U** des règles = tous les cas possibles.
- ∩ =∅

→ *L'algorithme ne doit pas oublier un cas, et on ne peut pas être dans deux cas à la fois.*

Quelques problèmes algorithmiques classiques

– **Le problème de tri** d'un ensemble d'éléments selon une relation d'ordre (croissant ou décroissant), avec un milliard d'éléments.
– **Le problème de la Recherche** pertinente dans des grandes bases de données.
– **Réalisation d'une Optimisation multi objectifs,** trouver une combinaison des meilleurs critères qui minimisent une fonction coût (temps de calcul et espace mémoire).

Qualités d'un bon algorithme

- **Correct:** Il faut que le programme exécute correctement les tâches pour lesquelles il a été conçu.
- **Complet**: Il faut que le programme considère tous les cas possibles et donne un résultat dans chaque cas.
- **Efficace**: Il faut que le programme exécute sa tâche avec efficacité (un coût minimal).

- Le **coût** pour un ordinateur se mesure en termes de temps de calcul et d'espace mémoire nécessaire.

Analyse et notions de complexité

- L'analyse d'algorithmes permet de donner un outil d'aide à **la comparaison des performances** des algorithmes.
- Le mot **complexité** (latin : ***complexus enlacé***, ***embrassé*** signifiant un objet constitué de plusieurs éléments ou plusieurs parties).

➢ Cet outil permet de mesurer les ressources critiques (**coûteuses**) utilisées par les algorithmes.
  - **Le temps d'exécution**
  - Les processeurs
  - **La mémoire**
  - Programmation
  - Les messages
  - Les implémentations matérielles d'un algorithme
  - L'énergie consommée

➢ La conception de certains ordinateurs mobiles permet de présenter le problème suivant :

- **Problème** : l'énergie pour la conception de certains ordinateurs mobiles (la consommation électrique) induite par l'émission réception.
  - **Solution** : évaluer les performances de l'algorithme en fonction de cette ressource,
    - ❖ La taille de cette ressource utilisée par l'algorithme.

Critères d'évaluation de performance de l'algorithme

- **L'efficacité** d'un algorithme est désignée sous le terme de « **complexité** ».
  - On la mesure, sur une taille d'entrée n donnée, par deux fonctions :
    - *Le nombre d'opérations* (complexité en temps)
    - *La place mémoire* (complexité en espace).

- **Définition.** La complexité est la mesure de **l'efficacité** d'un programme pour un type de ressources :
  — **Complexité temporelle** (temps de calcul CPU) : nombre d'opérations élémentaires réalisés par le processeur, ce qui est lié directement au temps de calcul.
  — **Complexité spatiale** (mémoire nécessaire) : espace mémoire RAM ou sur disque dur requis par le calcul.

Calcul de nombre d'opérations

Plusieurs façons d'écrire la performance d'un algorithme, avec des méthodes plus ou moins efficaces:

Unité de mesure = L'opération élémentaire, propre à chaque algorithme

En additionnant toutes les opérations élémentaires

→ une bonne estimation du coût (**coût est faible**).

→ **algorithme meilleur**

NB. La performance de l'algorithme dépend de ses entrées.

Opérations élémentaires

- L'exécution d'un algorithme est une séquence d'opérations nécessitant plus ou moins de temps.

- Pour mesurer ce temps, il faut préciser quelles sont les opérations élémentaires à prendre en compte.
  - **Pour un algorithme numérique**: les opérations arithmétiques de base +, *, - et / .
  - **Pour un algorithme de tri:** les comparaisons entre éléments à trier (opérations plus coûteuses). lire ou modifier un élément d'un tableau, ajouter un élément à la fin d'un tableau, affecter un entier ou un flottant.

    NB. Les transferts de valeurs ou affectations sont souvent négligées.

Mesure d'espace mémoire

C'est une ressource qui peut-être rare ou dont l'utilisation peut s'avérer très coûteuse du point de vue énergétique.

- Imaginons les satellites.

- Pour un problème donné, on peut donc être amené à choisir entre un algorithme rapide mais utilisant beaucoup de mémoire, et un algorithme plus lent qui utilise la mémoire de façon modérée.

Coût d'un algorithme

- **Définition 1.**

L'algorithme a un coût qui est lié :

- **au nombre d'opérations effectuées :** opérations arithmétiques, logiques, transferts,
- **à l'espace mémoire occupé par les données :** évaluer ce coût revient à mesurer ce qu'on appelle la complexité de l'algorithme, en temps comme en espace.
  - Il s'agit donc de *dénombrer* des opérations et des octets.

- **Définition 2.**

Coût de A sur n: *l'exécution de l'algorithme A sur la donnée n* requiert $C_A(n)$ opérations élémentaires.

- Estimer le coût d'une fonction sur une entrée de taille donnée signifie estimer le nombre de ces opérations élémentaires effectuées par la fonction sur l'entrée.

  - La complexité en mémoire consiste à estimer la mémoire nécessaire à une fonction pour son exécution.

Complexité d’un algorithme

- **Définition 3.**

On appelle complexité de l'algorithme A pour la ressource R la fonction :

T(A,R, n) = **max**{ **ressource R** utilisée par **A** sur l'ensemble des données **de taille n**}

- **Définition 4.**

La taille d'une donnée est simplement la taille d'un bon codage en mémoire de cette donnée exprimée en nombres de bits.

- La taille d'un codage d'un entier n sera : [(log2(n + 1)], et non pas n.
- ∄ d'ambiguïté, T(A,R, n)= T(n).

➢ La mesure du plus mauvais cas, car elle garantit que tout comportement de l'algorithme A utilisera ***au plus*** **T(A,R, n)** éléments de la ressource R.

Complexité d’un problème

- L’analyse d’un algorithme :
  - Exprimer quelques chose sur le problème associé.
  - La résolution d’un problème → tout algorithme n'a pas une complexité minimale.
    ≡ C'est la questions des *bornes inférieures* de complexité.

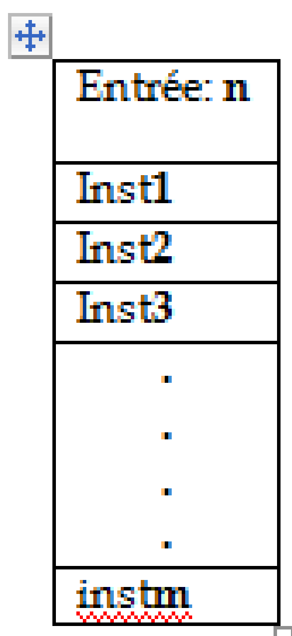
Problème : les calculs des fonctions de complexité sont difficiles à faire de manière exacte.

Solution : procède par approximations des notations **(O, Ω et Θ).**

Complexité d’un algorithme

- **Comment calculer la complexité d'un algorithme?**

Algorithme : problème



(a) **Ressources** : nombre d'instructions; **temporelle**

(b) **Paramètre n** C(n)

(c) **Asymptotique** ↗**n** **O(n)**

(d) **Pire des cas**

$C(n)=n^2+bn+c$
$O(n^2)$

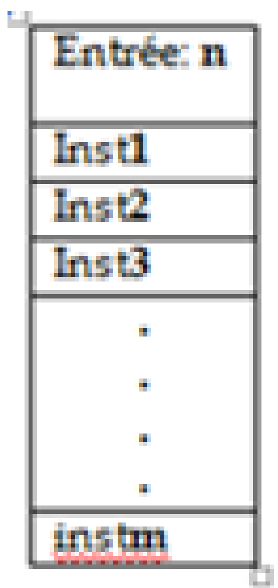
$C(n)=2^n+bn^2+c$
$O(2^n)$

Complexité d’un algorithme

- **Comment calculer la complexité d'un algorithme?**

Algorithme: Problème

Complexité?



$C(n+1)=C(n)$	$O(1)$
$C(n+1)=C(n)+1$	$O(n)$
$C(n+1)=C(n)+\varepsilon$	$O(\log_2(n))$
$2n \quad C(n)+1$	
$C(n+1)=C(n)+n$	$O(n^2)$
$C(n+1)=2*C(n)$	$O(2^n)$

❑ Type de complexité

▪ Complexité en temps d'un algorithme

Définition.

I : l'ensemble des données d'instances d'un problème abstrait Π.
In : les données de taille n (le coût dépend de la donnée)
c(i) : le coût de l'algorithme résolvant le problème Π pour une donnée i∈In.

On définit 3 types de complexité :

→ Le coût dans le pire des cas (l'algorithme est le moins performant) :

$$W_{algorithme}(n) = \max\{c(i) \mid i \in In\}$$

→ Le coût dans le meilleur des cas (l'algorithme est le plus performant) :

$$B_{algorithme}(n) = \min\{c(i) \mid i \in In\}$$

→ Le coût dans cas moyen :

$$A_{algorithme} = \sum_{i \in In} p(i) * c(i)$$

▪ Complexité en espace d'un algorithme

Permet de définir l'espace mémoire requis par le calcul.

❑ Notion d'optimalité

- **Définition.**

Un algorithme est dit **optimal** (**performant**) si sa complexité est ***la complexité minimale.***

➢ ***La complexité minimale*** **=** ***La borne inférieure*** de complexité parmi les algorithmes de sa **classe**.

PARTIE 2. GRANDEUR DES FONCTIONS, LA COMPLEXITÉ ASYMPTOTIQUE ET CALCUL DE COMPLEXITÉ DES ALGORITHMES ITÉRATIFS ET RÉCURSIFS

❑ Structures de données

- Une structure de données indique la manière d'organisation des données dans la mémoire.
- Le choix d'une structure de données adéquate dépend généralement du problème à résoudre.
- Deux types de structures de données :

❖ **Statiques** : Les données peuvent être manipulées dans la mémoire dans un espace statique alloué dès le début de résolution du problème.

❖ **Dynamiques** : On peut allouer de la mémoire pour y stocker des données au fur et à mesure des besoins de la résolution du problème.

❑ Qu'est ce que la récursivité ?

Définition.
Lorsqu'un système contient une autoréférence (ou une copie de lui même), on dit que ce système est récursif.

Une fonction récursive est caractérisée par une, ou plusieurs, ***relations de récurrence.***
La fonction récursive se rappelle jusqu'à arriver sur :

→ Cas d'arrêt (cas de base),

→ Quand on construit des fonctions récursives, le raisonnement adopté est en général:

- Comment traiter le cas le plus simple ?
- Comment me ramener d'un cas plus compliqué au cas simple ?

Exemple : La fonction factorielle
$n > 1 \rightarrow f(n) = n*f(n-1)$
$n = 0 \rightarrow f(n) = 1$
Le cas de base est $f(0) = 1$.

❑ Types de récursivité

- **Récursivité simple**

  Une récursivité simple contient un seul appel récursif à P dans le corps d’une procédure récursive P.

- **Récursivité multiple**

  Une récursivité est multiple si il y a plusieurs appels récursifs à P dans le corps d’une procédure récursive P.

- **Récursivité mutuelle**

  Une récursivité est mutuelle ou croisée quand une procédure P appelle une procédure Q qui déclenche un appel récursif à P.

- **Récursivité imbriquée**

  Une récursivité est imbriquée si une procédure récursive P contient un appel imbriqué.

- **Récursivité terminale**

❖ La récursivité est terminale si l’appel récursif est la dernière instruction et elle est isolée.

❑ Critères de terminaison pour un bon algorithme récursif

Règle 1: Un algorithme récursif doit être défini par une expression conditionnelle dont l'un au moins des cas mène à une expression évaluable sans appel récursif (condition d'arrêt).

Règle 2: Il faut s'assurer que pour toute valeur du (ou des) paramètre(s), il suffira d'un nombre fini d'appels récursifs pour atteindre la condition d'arrêt.

❑ Comment gérer la récursivité ?

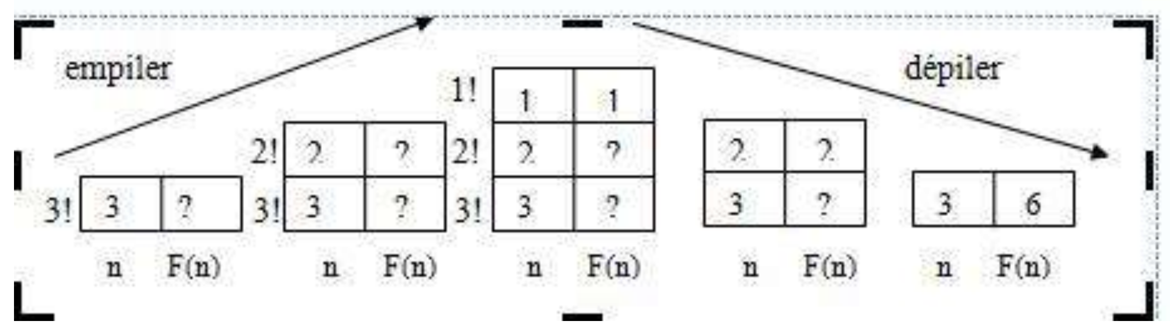
Quand la fonction est récursive, l'ordinateur est bien obligé de *stocker les calculs quelque part*. Il utilise pour cela une pile; il s'agit d'un objet défini par deux opérations :

- **Empiler** (ajouter un élément au sommet),
- **Dépiler** (prendre l'élément qui est au sommet).

Exemple :

1. Gestion de la pile pour la factorielle

Algorithme pour le calcul de la factorielle :
F(n) = F(m) (toujours une initialisation)
n > 1 → F(n) = n*F(n-1)
n = 0 → F(0) = 1



2. Une factorielle qui n'a pas besoin de pile

Algorithme pour la factorielle utilisant une variable acc, qui accumule :

F(n,acc) = F(m, 1)
n > 1 → F(n, acc) = F(n-1, n*acc)
n = 1 → F(1, acc) = acc

➢ Il n'y a pas besoin ici de mettre des calculs en attente, on peut les enchaîner.

❑ Gestion de la récursivité

➢ Gestion de la récursivité sans la pile : Récursivité Terminale (RT)

- La RT est une notion qui peut améliorer nettement les performances de vos algorithmes.
- L'exécution d'un algorithme utilisant la RT est transformée en général en algorithme itératif
  - ❖ Plus rapide et moins gourmand en mémoire par le compilateur.

Définition.
Une fonction est RT si elle renvoie, sans autre calcul, la valeur obtenue par son appel récursif.

❖ **F(x1, ..., xn) = G(y1, ..., yn)** ➡ **Pas besoin de pile** pour continuer le calcul.

➢ Gestion de la récursivité avec la pile : Récursivité Enveloppée (RE)

Définition.
Une fonction est de type RE, lorsqu'il y a quelque chose autour de la fonction G,

❖ Il y a des opérations qui enveloppent la fonction ➡ **Besoin de pile** pour continuer le calcul.

Optimisation d'exécution de la récursivité

- Optimisation d'exécution de la récursivité par la transformation de RE vers RT,
  - En remplaçant son enregistrement au lieu d'en empiler un autre au-dessus de lui,
  - On doit donc transformer les fonctions récursives en fonctions récursives terminales à chaque fois que **cela est possible**.

- Pour optimiser l'exécution, **les langages** exécutent un programme à
  - RT comme s'il était itératif ≡ en espace constant.
  - Sinon, il est facile de transformer une définition récursive terminale en itération.

Comparaison d'algorithmes

- **Gestion de la récursivité pour le calcul de la factorielle :**

- La **RT (preuve):**

  F(n, acc) = F(n-1, n*acc), On peut identifier $y_1 = x_1 - 1$, $y_2 = n*x_2$ et G = F.
  F(1, acc) = acc. On peut identifier $y_1 = 0$, $y_2 = x_2$ et $G = y_2$.

- La **RE (preuve):**

  F(n) = n*F(n-1), La fonction est enveloppée par une multiplication avec n.

Gestion de la récursivité pour faire la somme des éléments d'un tableau

- On dispose d'un tableau T d'entiers, et on veut faire la somme de tous ses éléments

❖ **La RE** : les règles trouvées

1. $\Sigma(T, n) = \Sigma(T, |T|)$
2. $n \geq 1 \rightarrow \Sigma(T, n) = T[n] + \Sigma(T, n-1)$
3. $n < 1 \rightarrow \Sigma(T, n) = 0$

Problème : On remarque à la ligne 2 que l'appel est enveloppé par l'addition avec T[n] :

- **RE → utiliser la pile.**

Solution : Traduire ceci en une **RT.**

- Le but maintenant est de faire **rentrer** l'addition avec T[n] dans la fonction.
- Utiliser **un accumulateur** : au départ on le met à 0, et à chaque appel on y ajoute T[n].
- A la fin, il suffira de rendre la valeur accumulée :

❖ **La RT** : les règles trouvées ≡ algorithme avec une boucle

1. $\Sigma(T, n, acc) = \Sigma(T, |T|, 0)$
2. $n \geq 1 \rightarrow \Sigma(T, n, acc) = \Sigma(T, n-1, acc + T[n])$
3. $n < 1 \rightarrow \Sigma(T, n, acc) = acc$

❖ Traduction d’un algorithme récursif vers un algorithme itératif

- Traduction d’un algorithme de RT vers un algorithme avec une boucle

```
fonction F(T : tableau[1..n] d’éléments entier; n, acc : entier) : entier
Début
  acc ← 0;
  tant que (n ≥ 1) faire
     debut
     acc ← acc + T[n];
     n ← n – 1;
     fin;
  retourner(acc) ;
fin.
```

❑ C'est quoi les notions de Landau ?

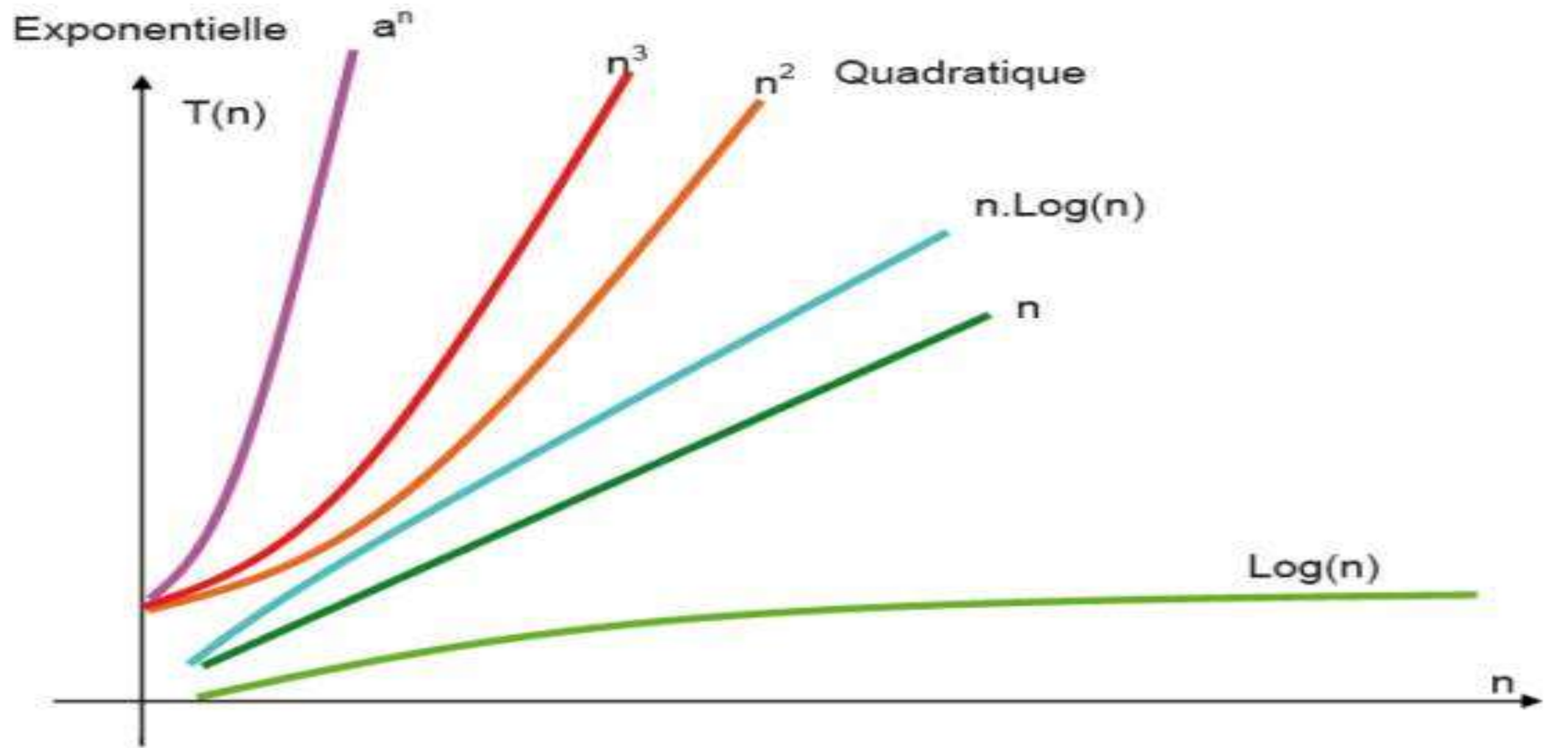
- La plupart des programmes contient un nombre d'instructions très élevés,

- L'évaluation de la complexité de façon exact pour comparer deux algorithmes n'est pas facile.
  - ❖ Parce que des critères matérielles sont introduite dans le calcul.

Classes de complexité

Complexités de référence	Notation	Fonction	Définition
Complexité Constante	$O(1)$	$f(n)=1$	Accéder au premier élément d'un ensemble de données.
Complexité Logarithmique	$O(logn)$	$f(n)=log(n)$	Couper un ensemble de données en deux parties égales, puis couper ces moitiés en deux parties égales.
Complexité Linéaire	$O(n)$	$f(n)=n$	Parcourir un ensemble de données.
Complexité Quasi-linéaire	$O(nlogn)$	$f(n)=nlog(n)$	Couper répétitivement un ensemble de données en deux et combiner les solutions partielles pour calculer la solution générale.
Complexité Quadratique	$O(n^2)$	$f(n)= n^2$	Parcourir un ensemble de données en utilisant deux boucles imbriquées.
Complexité Cubique	$O(n^3)$	$f(n)= n^3$	Parcourir un ensemble de données en utilisant trois boucles imbriquées.
Complexité Polynomiale	$O(n^P)$	$f(n)= n^p$	Parcourir un ensemble de données en utilisant P boucles imbriquées.
Complexité Exponentielle	$O(2^n)$	$f(n)= 2^n$	Générer tous les sous-ensembles possibles d'un ensemble de données.

❑ Classes de complexité



❑ Notations asymptotique

➢ **Domination asymptotique**

- Quand la complexité g(n) de l'algorithme est **majorée** par f(n), on dit qu'il est en **O(f(n)).**

$$O(f(n)) = \{ g, \exists n0, \exists c>0, \forall n>n0, g(n) \leq c.f(n) \}$$

➢ **Notation Ω**

- Quand la complexité g(n) de l'algorithme est **minorée** par f(n), on dit qu'il est en **Ω(f(n)).**

$$\Omega(f(n)) = \{ g, \exists n0, \exists c>0, \forall n>n0, g(n) \geq c.f(n) \}$$

➢ **Equivalence asymptotique**

-Quand la complexité f(n) de l'algorithme est **exactement** en g(n), on dit qu'il est en **Θ(g(n)).**

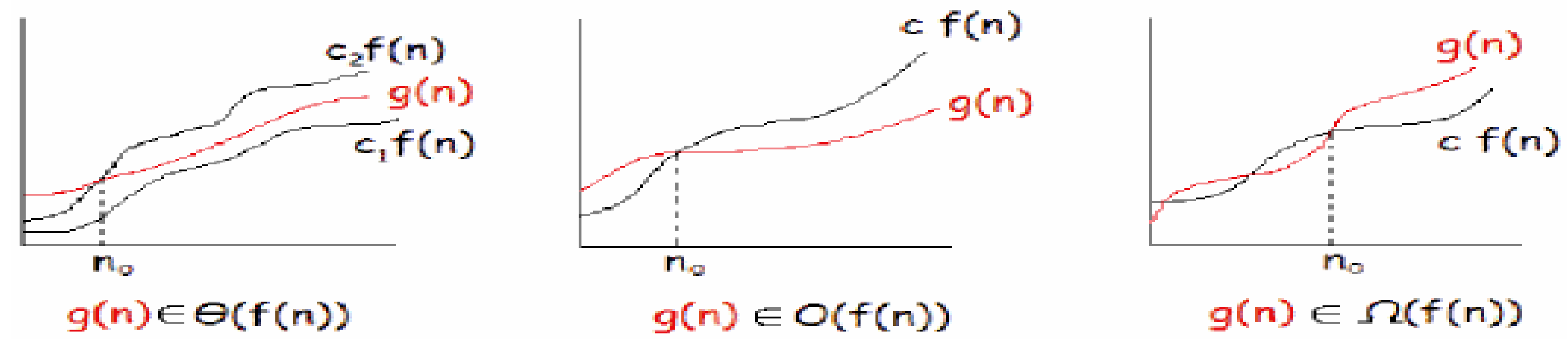


Figure. **Les notations de Landau**

Calcul de complexité asymptotique

- Nous concentrons sur **le coût** des **actions** résultant de l'exécution de l'algorithme,
  - En fonction d'une **"taille" n** des données traitées.
  - En fonction de **la nature de n**.

- Ceci permet en particulier de comparer deux algorithmes traitant le même calcul.

❑ Calcul de complexité asymptotique

- 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.
  - ❖ Le temps d'exécution dépend de **la nature** des données.
- **Définition**

❖ Borne supérieure non asymptotiquement approchée (négligeable).

❖ g(n) est une borne supérieure non **asymtotiquement** approchée de f(n) ou que f est négligeable devant g.

Si $\lim_{n\to\infty} f(n)/g(n) = 0$ alors On note **f(n) =O(g(n))**

Calcul de complexité asymptotique

Exemple :

```
{début}
K  0 ;
I  1 ;
{#1}
TantQue I ≤ N {#2} Faire
    R  R+T[I]; {#3}
    I  I+1;{#4}
FinTantQue;
{fin}.
```

Le temps d'exécution t(n) de cet algorithme en supposant que:

- N=n
- t1 est le temps d'exécution entre le début et {#1}
- t2 est le temps d'exécution de la comparaison {#2}
- t3 est le temps d'exécution de l'action {#3}
- t4 est le temps d'exécution de l'action {#4}
- t1, t2, t3, t4 sont des constantes (ne dépendent pas de n)

Calcul de complexité asymptotique

- $t(n)= t1 + \sum_{i=1}^{n} (t2+t3+t4) + t2$

et en définissant le temps **tit** d'exécution d'une itération (condition comprise),

$tit= (t2+t3+t4)$ d'où

$t(n)= t1+ t2+n \times tit$

- Ce qui signifie que le temps d'exécution dépend linéairement (plus précisément est une fonction affine) de la taille n.

- Nous intéressions au comportement asymptotique:

$\lim_{n \to \infty} t(n)/tit \times n =1$ autrement dit ,

t(n) **est équivalent à l'infini à tit*n : t(n) $\sim_{\infty}$ tit×n**

- L'algorithme est donc **asymptotiquement linéaire en n.**

NB. Dans cet exemple simple l'évaluation en "**pire des cas**" est immédiate puisque t(n) ne dépend pas de la nature des données,

Exercice

Le but de cet exercice est de tester votre compréhension de la notion de complexité dans le pire des cas.

1. Si je prouve que la complexité dans le pire des cas d'un algorithme est en $O(n^2)$, est-il possible qu'il soit en $O(n)$ sur certaines données?
2. Si je prouve que la complexité dans le pire des cas d'un algorithme est en $O(n^2)$, est-il possible qu'il soit en $O(n)$ sur toutes les données?
3. Si je prouve que la complexité dans le pire des cas d'un algorithme est en $\Theta(n^2)$, est-il possible qu'il soit en $O(n)$ sur certaines données?
4. Si je prouve que la complexité dans le pire des cas d'un algorithme est en $\Theta(n^2)$, est-il possible qu'il soit en $O(n)$ sur toutes les données

Exercice

Le but de cet exercice est de tester votre compréhension de la notion de complexité dans le pire des cas.

1. Si je prouve que la complexité dans le pire des cas d'un algorithme est en $O(n^2)$, est-il possible qu'il soit en $O(n)$ sur certaines données?
2. Si je prouve que la complexité dans le pire des cas d'un algorithme est en $O(n^2)$, est-il possible qu'il soit en $O(n)$ sur toutes les données?
3. Si je prouve que la complexité dans le pire des cas d'un algorithme est en $\Theta(n^2)$, est-il possible qu'il soit en $O(n)$ sur certaines données?
4. Si je prouve que la complexité dans le pire des cas d'un algorithme est en $\Theta(n^2)$, est-il possible qu'il soit en $O(n)$ sur toutes les données

Solution:

1. OUI
2. OUI
3. OUI
4. NON

PARTIE 2. CALCUL DE COMPLEXITÉ DES ALGORITHMES ITÉRATIFS ET RÉCURSIFS

❑ A propos de la notation O

❖ Cette notation fait abstraction des détails liés à la machine mais aussi de certaines opérations de programmation.

Exemple : comment vérifier qu'un élément est dans un tableau.

```
Fonction test(T : tableau [1..n]d'élément entier ;x,n,i : entier) :booléen
Début
   i←1 ;
   Tant que  (i ≤ n) et (T[i] ≠ x) alors
            i←i+1;
   fintant que;
   si (i ≤ n) alors
     retourner(vrai) ;
   sinon
     retourner(faux) ;
   finsi ;
Fin.
```

→ La complexité est en O(n).

- Si on ajoute x à la fin du tableau,

```
Fonction test(T : tableau [1..n]d'élément entier ;x,n,i : entier) :booléen
Début
   i←1 ;
  si (T[n] = x) alors retourner(vrai) ;
  sinon
      T[n] ← x;
  finsi;
 Tant que  (i ≤ n) et (T[i] ≠ x) alors
            i←i+1;
  fintant que;
  si (i ≤ n) alors
     retourner(vrai) ;
  sinon
     retourner(faux) ;
  finsi ;
Fin.
```

→ La complexité est en O(n).

- Les **opérations de programmation** qui peuvent avoir une certaine importance disparaissent avec cette notation **asymptotique**.

- **Problème** :

  - **O** est un “**pire des cas**” : il n’indique donc pas que l’algorithme va prendre exactement ce temps, mais qu’il ne peut pas prendre davantage.

  - La notation **O** fait disparaître quelques constantes liées à la programmation dont les tests dans les boucles et de petites optimisations.

CALCUL DE COMPLEXITÉ DES ALGORITHMES ITÉRATIFS

- **Etape 1 :** Sous quel aspect doit-on considérer la donnée d'entrée ?
  - Soit comme le nombre d'objets donnés → **analyse uniforme** (classique).
  - Soit comme la taille des bits nécessaires pour la représentation → **analyse logarithmique**.

  Pour un nombre x, il faut $\lceil \log_2(x+1) \rceil$ bits, ou $\lfloor \log_2(x) \rfloor + 1$

- **Etape 2 :** Quelles sont les opérations élémentaires, i.e. celles qui se font en temps constant ?
  - En analyse uniforme → Opérations arithmétiques **+, -, *, /, %.**
  - En analyse logarithmique → Manipulation de bits : décalage, test à zéro, additions de bits.

CALCUL DE COMPLEXITÉ DES ALGORITHMES ITÉRATIFS

- **Etape 3 :** On se donne une base de données de complexité élémentaires et des règles sur les opérations.
  - **w(i, J) = w(i) + w(J).** Le coût d'instructions séquentielles est le coût de l'une plus celui de l'autre.
  - **w(x ← e) = cost(e).** Le coût d'une affectation est le coût du calcul/manipulation de la donnée.
  - **w(if(c) S_1 else S_2) = w(c) + max(w(S_1), w(S_2)).** On paye la condition puis le pire des cas.
  - **w(while (c), S) = $\Sigma_{itérations}$(w(c) + w(S)).** On paye le test et les opérations à chaque passage.
  - **w(for(S1 ; S2 ; S3), S4) = w(S1) + Σitérations(w(S2) + w(S3) + w(S4)).**
  - **w(récursivité)** : coût de la récursivité.

CALCUL DE COMPLEXITÉ DES ALGORITHMES RÉCURSIFS

- La complexité de l'algorithme récursif est donnée par son équation,
- La résoudre en utilisant un formulaire pour les cas standard.
- Le coût de la récursivité est donné par T(m) tel que :

$T(m) = \alpha$ cas de base

$T(m) = c.T(f(m)) + g(m)$ équation de récurrence

Les paramètres sont :

- **α** : temps d'exécution dans le cas de base. En général **O(1).**
- **c** : nombre de fois qu'on fait un appel récursif.

Exemple : diviser pour régner

Deux appels récursifs (sur les éléments de la première moitié, et de la seconde).

- g(m), le coût d'un appel récursif
- f(m), la taille de la donnée m lors de l'appel récursif,

  $f(m) = m - 1$ ou

  $f(m) = m - 2$.

Formulaire des solutions d'équations de récurrence où **T(1) = O(1)** :

- $T(n) = T(n/2) + O(1)$ → $T(n) = O(\log n)$
- $T(n) = T(n-1) + O(\log n)$ → $T(n) = O(n.\log n)$
- $T(n) = c.T(n-1) + O(n^k)$ → $T(n) = O(c^k)$
- $T(n) = c.T(n/d) + O(n^k)$ → si $c = d^k$, alors $T(n) = O(n^k.\log n)$
  → si $c < d^k$, alors $T(n) = O(n^k)$
  → si $c > d^k$, alors $T(n) = O(n^{\log \text{ base } d \text{ de } c})$

ÉQUATIONS DE RÉCURRENCES

- **Théorème 1.** Les équations de récurrence :

$$\begin{cases} T(1) = c, \\ T(n) = aT(n/b) + cn, \quad n \geq 2 \end{cases}$$

avec $a,b,c > 0$ et où n/b représente soit [n/b] soit [n/b], ont pour solution :

- ❖ $a < b \Rightarrow T(n) \in \Theta(n)$
- ❖ $a = b \Rightarrow T(n) \in \Theta(n\log(n))$
- ❖ $a > b \Rightarrow T(n) \in \Theta(n^{\log_b(a)})$

.

- **<u>Théorème 2</u>.** Les équations de récurrence :

$$\begin{cases} T(1) = c \\ T(n) = aT(n/b) + c, \quad n \geq 2 \end{cases}$$

avec $a \geq 1, b > 1, c > 0$ et où n/b représente soit [n/b] soit [n/b] ont pour solution :

- ❖ $a < b$ si $a = 1 \Rightarrow T(n) \;\; \Theta(\log(n))$
  si $a > 1 \Rightarrow T(n) \;\; \Theta(n^{\log_b(a)})$
- ❖ $a = b \Rightarrow T(n) \;\; \Theta(n)$
- ❖ $a > b \Rightarrow T(n) \;\; \Theta(n^{\log_b(a)})$

Remarque 1. Le cas où $a=1$ et $b=2$ correspond au processus dichotomique classique. Pour démontrer les deux théorèmes précédents, on peut commencer par considérer les entiers $n = b^t$.

- **<u>Théorème 3</u>.** Les équations de récurrence :

$$\begin{cases} T(1) = c \\ T(n) = aT(n/b) + f(n) \text{ , } n \geq 2 \end{cases}$$

avec $a \geq 1, b > 1$ et où n/b représente soit $[n/b]$ soit $[n/b]$ ont pour solution :

- ❖ si $f(n) \in O(n^{\log_b(a)-\varepsilon})$ pour $> 0 \Rightarrow T(n) \in \Theta(n^{\log_b(a)})$
- ❖ si $f(n) \in \Theta(n^{\log_b(a)}) \Rightarrow T(n) \in \Theta(n^{\log_b(a)}\log(n)) = \Theta(f(n)\log(n))$
- ❖ si $f(n) \in \Omega(n^{\log_b(a)+\varepsilon})$ pour > 0 et si $af(n/b) \leq df(n)$ avec $d < 1, \Rightarrow T(n) \in \Theta(f(n))$

Remarque 2. Ce théorème ne couvre pas tous les cas possibles de la fonction f. Enfin terminons par les équations d'autres schémas récursifs.

- **Théorème 4.** Les équations de récurrence :

$$\begin{cases} T(1) = a \\ T(n) = bT(n-1) + a, n \geq 2 \end{cases}$$

avec $a > 0, b \geq 1$ ont pour solution :

- ❖ $b = 1 \Rightarrow T(n) = an \in \Theta(n)$
- ❖ $b \geq 1 \Rightarrow T(n) \in \Theta(b^{n-1})$

Remarque 3. Le problème des tours de Hanoï correspond au cas où $a = 1, b = 2$, et $T(n) \in \Theta(2^{n-1})$, donc un algorithme récursif exponentiel.

- **Théorème 5.** Les équations de récurrence :

$$\begin{cases} T(1) = a \\ T(n) = bT(n-1) + an \text{ , } n \geq 2 \end{cases}$$

avec $a > 0$, $b \geq 1$ ont pour solution :

- $b = 1 \Rightarrow T(n) \in \Theta(n^2)$
- $b \geq 1 \Rightarrow T(n) \in \Theta(b^{n-1})$

Remarque 4. Le plus mauvais cas de l'algorithme Quicksort correspond au cas où $b = 1$, $a = 1$, cet algorithme a donc un comportement quadratique dans le pire des cas.

Propriétés.

1. $T(1) = a$
   $T(n) = T(n/5) + T(3n/10) + an, n \geq 2$ $\Rightarrow$ $T(n) \in O(n)$

2. $T(1) = a$
   $T(n) =) + an, n \geq 2$ $\Rightarrow$ si $\sum_{i=1}^{i=k} \frac{ai}{bi} < 1$ alors $T(n) \in O(n)$

Exercice –

1. Pour résoudre un problème manipulant un nombre important de données, on propose deux algorithmes :
   a. un algorithme qui résout un problème de taille n en le divisant en 2 sous-problèmes de taille n/2 et qui combine les solutions en temps quadratique,
   b. un algorithme qui résout un problème de taille n en le divisant en 4 sous-problèmes de taille n/2 et qui combine les solutions en temps $O(\sqrt{n})$. Lequel faut-il choisir?
2. Même question avec les algorithmes suivants :
   a. un algorithme qui résout un problème de taille n en le réduisant à un sous-problème de taille n/2 et qui combine les solutions en temps $\Omega(\sqrt{n})$,
   b. un algorithme qui résout un problème de taille n en le divisant en 2 sous-problèmes de taille n/2 et qui combine les solutions en temps constant.

Solution:

1. **a.** $T(n) = 2T\left(\frac{n}{2}\right) + n^2 \Rightarrow T(n) = \Theta(n^2)$.

   b. $a = 4$, $b = 2$, $f(n) = O(\sqrt{n})$, cas 1. $\varepsilon = \frac{3}{2}$, $\Rightarrow T(n) = \Theta(n^2)$.

   Les 2 algos sont équivalents.

2. **a.** $a = 1$, $b = 2$, $f(n) = \Omega(n^{\frac{1}{2}})$, cas 3. $\varepsilon = \frac{1}{2}$, $c = 1/\sqrt{2}$ par exemple. $\Rightarrow T(n) = \Theta(\sqrt{n})$.

   b. $a = b = 2$, $f(n) = \Theta(1)$, cas 1. $\varepsilon = 1$, $\Rightarrow T(n) = \Theta(n)$.

   Le premier algo est meilleur.