Présentation Générale :

Module : Complexité Algorithmique
21H de cours / 10h30 de TD

Objectif principal: Apprendre à évaluer les algorithmes et à proposer des algorithmes efficaces et optimisés

Enseignant : Mme Sajeh ZAIRI
Docteur-Ingénieur
Maître-Assistante à l’ESEN
sajeh.zairi@esen.tn

PLAN DU COURS

- Chapitre 1: Notions et concepts de base
- Chapitre 2: Complexité d’algorithmes itératifs
- Chapitre 3: Complexité d’algorithmes récursifs

Chapitre 1

NOTIONS ET CONCEPTS DE BASE

Exemple Introductif

- Soit à calculer l'expression suivante :
  $E = 1 + x + x^2 + x^3 + \ldots\ldots\ldots x^{n-1} + x^n$
  avec x un réel > 0 et n un entier > 2

- Ecrire une fonction puissance (x,i) qui calcule x^i

et calculer le nombre d'opérations arithmétiques

- Proposer un algorithme $\mathcal{A}1$ qui calcule E et calculer le nombre des opérations
- Peut-on proposer un algorithme $\mathcal{A}2$ meilleur que $\mathcal{A}1$?

Exemple Introductif

```
puissance  (x, i)
Debut
p := x
pour j de 2 à i faire
    p := p*x
fin pour
puissance :=p
Fin
```

i-1 (*)

AlgoA1

```
Début
E := 1
pour i de 1 à n faire
    E := E+ puissance (x,i)
fin pour
Fin
```

n (+)

1 + 2 + ……… + (n-1) (*)

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

$=> \frac{n(n-1)}{2} + n$

$= \frac{n(n+1)}{2}$ opérations (o.a)

pour n=1000, on fait presque 500000 o.a

Exemple Introductif

Algo A2

Debut

E := 1 , p := x

pour i de 2 à n faire

E := E + p

p := p * x

fin pour

E := E + p

Fin

On peut se limiter à

n-1 (*)

n (+)

=> *2n-1* opérations

pour n=1000, on fait 1999 o.a

Désormais, *A1* est inacceptable de la part d'un informaticien.
Penser aux performances de l'algorithme est une nécessité !

En conclusion,

Résoudre un problème est une chose, le résoudre efficacement en est une autre!

Importance de l'évaluation des performances (efficacité) des algorithmes dite complexité algorithmique

Problème et algorithmes

- $\mathcal{P}$(n) : un problème à résoudre où n est la taille du problème dit paramètre du problème
- $\mathcal{A}$(n) : algorithme permettant la résolution de $\mathcal{P}$(n)

- $\mathcal{A}$(n) = application entre 2 ensembles
  - Ensemble d'entrée E(n)
  - Ensemble de sortie S(n)
- Plusieurs algorithmes $\mathcal{A}i$(n) i=1.. peuvent résoudre le même problème $\mathcal{P}$(n)

Problème et algorithmes-Exemple

- $\mathcal{P}$(n) : un problème de tri d'un tableau de taille n
- (n est le paramètre du problème)
- $\mathcal{A}$(n) : algorithme de tri par sélection

Entrée et sortie de $\mathcal{A}$(n)

- – Ensemble d'entrée E(n) : Le tableau à trier; sa taille
- – Ensemble de sortie S(n) : Le tableau trié

- Plusieurs algorithmes $\mathcal{A}i$(n) peuvent résoudre le même problème $\mathcal{P}$(n), exemple tri par insertion, tri à bulles, etc.

Problématique de l'algorithmique

- La question fréquemment posée par un programmeur est :
  - **Comment choisir parmi les différents algorithmes pour résoudre un problème?**
- Un algorithme doit résoudre correctement et de manière efficace le problème
- L'efficacité : en combien de temps et avec quelles ressources ?
- Plutôt souhaitable que nos solutions
  - ne consomment pas beaucoup de temps (rapide)
  - ne consomment pas un espace mémoire considérable

Définitions : Complexité

- ***La théorie de la complexité étudie l'efficacité des algorithmes***
  - La complexité algorithmique est une métrique (mesure) de **l'évaluation des performances des algorithmes**

- L'efficacité d'un algorithme est évaluée par :
  - Rapidité (en terme de temps d'exécution) **: complexité temporelle**
  - Consommation de ressources (espace de stockage, mémoire utilisée) **: complexité spatiale**

Il est Primordial de considérer l'optimisation de la complexité au moment de la proposition d'un algorithme

Définitions : Complexité temporelle

Estimation du temps d'exécution par le calcul du nombre d'opérations élémentaires effectuées par l'algorithme => Utilisation d'une unité de temps abstraite

- La durée d'exécution n'est donc pas mesurée en heure, minute, seconde mais en unité de temps
- Il ne s'agit pas d'implémenter les algorithmes qu'on veut comparer. La comparaison se fait au préalable, bien avant l'étape d'implémentation
- Même si on implémente pour mesurer le temps, les mesures seront impertinentes
  - On aura des valeurs numériques qui ne varient pas en fonction de la taille du problème
  - On ne saura pas distinguer entre le temps de calcul et le temps du aux transferts mémoires et aux entrées sorties
  - Le même algorithme sera plus rapide sur une machine plus puissante

Calcul de la complexité temporelle

- On distingue :
  - Les opérations élémentaires faites par les processeurs :
    - Opérations arithmétiques : + - * et /
    - Opérations logiques : ET, OU, … et comparaisons
  - Les instructions qui consistent en un transfert de données entre processeur et mémoire:
    - lectures, écritures et affectations

- Un ensemble de conventions (propre à nous dans ce cours)
  - Chaque **opération élémentaire** coûte une unité de temps (même si en réalité elles n'ont pas toutes de même coût)
  - **Les affectations, lectures et écritures** sont négligeables devant les opérations arithmétiques et logiques, mais on doit bien sûr les considérer si ces dernières n'existent pas ou sont très peu dans l'algorithme
  - **Dans une boucle**, on néglige le temps de l'incrémentation du compteur devant celui des instructions du corps de la boucle
  - Chaque **appel de fonction** rajoute le nombre d'unités de temps consommées dans cette fonction, mais pas de temps attribué à l'appel lui-même

Définitions : Complexité spatiale

- **Estimation de l'espace mémoire par le nombre de variables utilisées par $\mathcal{A}$(n) et de leurs types**

- Vu l'évolution actuelles des tailles des mémoires, le problème de la complexité spatiale se pose de moins en moins

- Par conséquent, un intérêt particulier est accordé à la complexité temporelle

C'est le cas pour la suite de ce cours

Notations

- $\mathscr{P}$(n) → $\mathscr{A}$(n) → $\mathscr{C}$(n)
- Avec :
  - $\mathscr{P}$: le problème à résoudre
  - n : paramètre du problème
  - $\mathscr{A}$: Algorithme qui résout $\mathscr{P}$
  - $\mathscr{C}$: complexité de $\mathscr{A}$

Exemple 2

Soit l'algorithme calculant n!   n! =1*2*…*(n-1)*n

```
factoriel(n)
Debut
   fact :=1
   pour i de 2 à n faire
        fact := fact * i    //  1 multiplication (*)
   finpour
   factoriel := fact
Fin
```

Complexité temporelle : $\mathcal{C}(n) = (n-1)$

Complexité spatiale : 3*(taille d'un entier)

Exemple 3

- Soient U, V et W des vecteurs de taille n de réels. Ecrire un algorithme calculant W=U+V et calculer sa complexité temporelle et spatiale.

pour i de 1 à n faire
W[i] := U[i] + V[i]
fin pour

- Complexité temporelle : n (+) => $\mathcal{C}$**(n) = n**

- Complexité spatiale : 3n*(taille d'un réel) + 2*(taille d'un entier)

Exemples d'application

- Pour chacune des descriptions suivantes :

1. Écrire l'algorithme correspondant
2. Donner la taille du problème
3. Donner l'opération à comptabiliser
4. Donner le nombre des opérations

Exemples d'application : Algo1

Algo1 : affiche les composantes d'un vecteur x ayant n composantes

pour i de 1 à n faire
Afficher (x[i])
fin pour

- La taille du problème est donc n.
- L'opération que l'on compte est Afficher (x[i])
- Le nombre d'opérations est donc n.

Exemples d’application : Algo2

Algo2 : affiche la somme de toutes les paires de deux vecteurs x, y ayant n composantes chacun

Exemples d'application : Algo2

Algo2 : affiche la somme de toutes les paires de deux vecteurs x, y ayant n composantes chacun

```
pour i de 1 à n faire
    pour j de 1 à n faire
        Afficher (x[i]+y[j])
    fin pour
fin pour
```

-

Exemples d’application : Algo2

Algo2 : affiche la somme de toutes les paires de deux vecteurs x, y ayant n composantes chacun

pour i de 1 à n faire
　　pour j de 1 à n faire
　　　　Afficher (x[i]+y[j])
　　fin pour
fin pour

- La taille du problème est donc n.
- L’opération que l’on compte est Afficher (x[i]+y[j])
- Le nombre d’opérations est donc n^2.

Exemples d’application : Algo3

Algo3 : affiche la somme de toutes les quintuplés (5 éléments) du vecteur x ayant n composantes

Exemples d’application : Algo3

Algo3 : affiche la somme de toutes les quintuplés (5 éléments) du vecteur x ayant n composantes

```
pour i de 1 à n faire
  pour j de 1 à n faire
    pour k de 1 à n faire
      pour l de 1 à n faire
        pour m de 1 à n faire
           Afficher (x[i]+x[j]+x[k]+x[l]+x[m])
        fin pour
      fin pour
    fin pour
  fin pour
fin pour
```

Exemples d'application : Algo3

Algo3 : affiche la somme de toutes les quintuplés (5 éléments) du vecteur x ayant n composantes

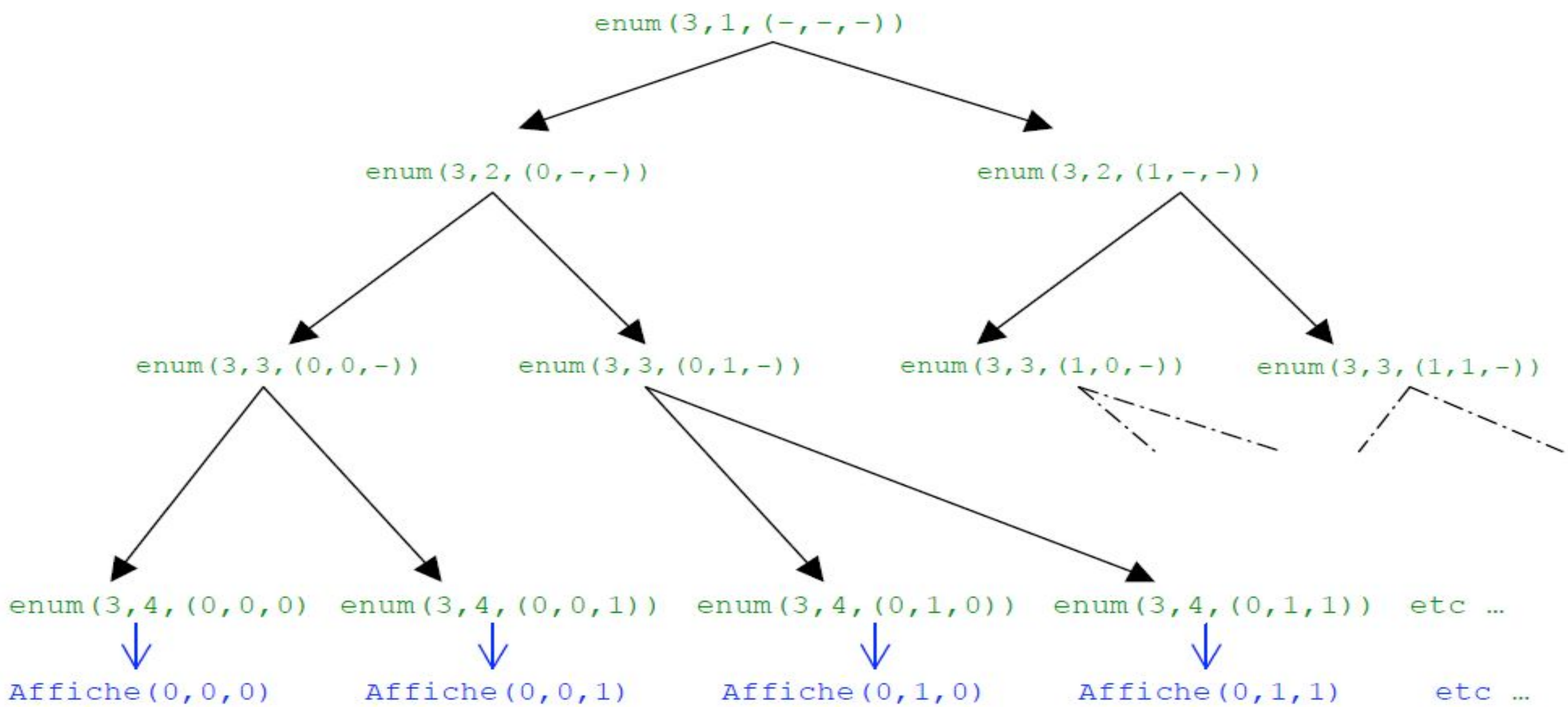
```
pour i de 1 à n faire
  pour j de 1 à n faire
    pour k de 1 à n faire
      pour l de 1 à n faire
        pour m de 1 à n faire
           Afficher (x[i]+x[j]+x[k]+x[l]+x[m])
        fin pour
      fin pour
    fin pour
  fin pour
fin pour
```

- La taille du problème est donc n
- L'opération que l'on compte est Afficher (x[i]+x[j]+x[k]+x[l]+x[m])
- Le nombre d'opérations est donc n^5

Exemples d'application : Algo4

Algo4 : affiche toutes les valeurs possibles des vecteurs de n composantes, toutes valant 0 ou 1.

- Indication: algo récursif.
- Voici l'arbre qui explique l'affichage pour n=3 : enum(3,1,x)



```
enum(3, 1, (-,-,-))
enum(3, 2, (0,-,-))
    enum(3, 3, (0,0,-))
        enum(3, 4, (0,0,0))
        enum(3, 4, (0,0,1))
    enum(3, 3, (0,1,-))
        enum(3, 4, (0,1,0))
        enum(3, 4, (0,1,1))
enum(3, 2, (1,-,-))
    enum(3, 3, (1,0,-))
    enum(3, 3, (1,1,-))
```

Exemples d'application : Algo4

```
enum (n,i,x)
Debut
    Si i> n alors
        Afficher (x)
    Sinon
        x[i]:=0
        enum (n,i+1,x)
        x[i]:=1
        enum (n,i+1,x)
    finsi
Fin
```

Premier appel:

enum (n,1,x)

- La taille du problème est n (car en entrée, on a un vecteur x de n composantes indéfinies)
- Ici, on considère que Afficher(x) est l'opération élémentaire qu'on compte. Cette opération est effectuée 2^n fois. On aurait pu considérer que l'opération à compter était l'affichage d'une composante de x. Dans ce cas Afficher(x) comporterait n opérations (cf. Algo1)
- donc Algo4 compterait $n \times 2^n$ opérations.

Ordre de complexité
Notations de Landau

- Quand on calcule la complexité d'un algorithme, on ne calcule généralement pas sa complexité exacte, mais son ordre de grandeur
- Car on s'intéresse à des données très grandes parce que les petites valeurs ne sont pas assez informatives (la taille du problème tend vers l'infini)

- <u>C'est une approximation du temps de calcul de l'algorithme</u>

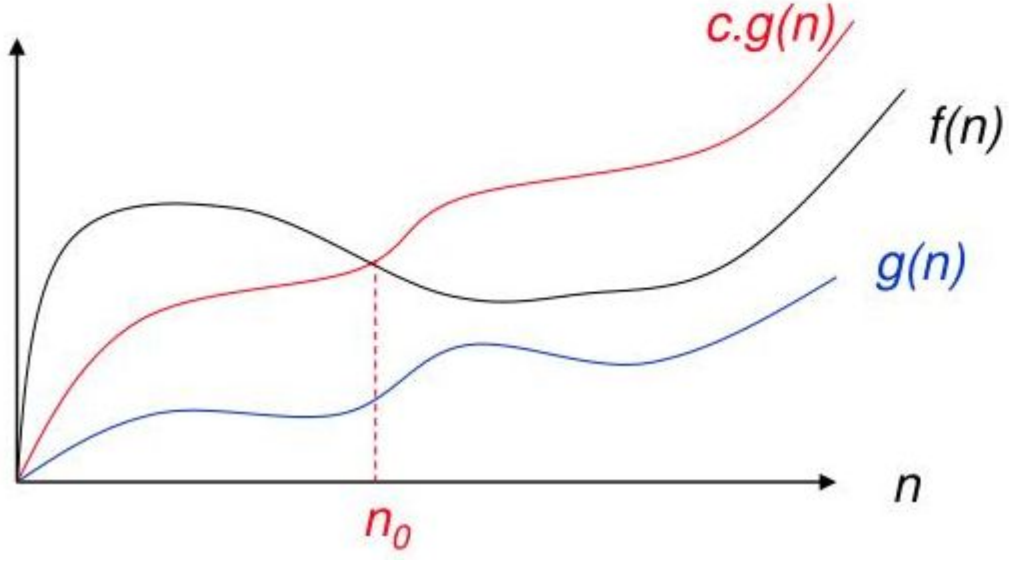
Par exemple : si on fait $4n^2 + 2n - 5$opérations, on retiendra juste $4n^2$ ou encore que l'ordre de grandeur est n^2

- Notations asymptotiques O (.) : NOTATIONS de LANDAU
- O (.) Ω(.) Θ (.)

Notation O

- Soit f(n) et g(n) deux fonctions
- $f(n) = O(g(n))$ ssi
  $\exists\, n_0 \in IN$ et $c > 0 \in IR$ tq $f(n) \leq c \,.\, g(n) \;\; \forall\; n \geq n0$
- f(n) est en O(g(n)) s'il existe un seuil à partir duquel la fonction f(.) est toujours dominée par g(.), à une constante multiplicative près
- Exemple
  - $f(n) = 3n^2 + 17n$
  - $g(n) = n^3$ $\quad\quad$ => $f(n) = O(g(n))$
- **Utilité :** Le temps d'exécution est borné (borne supérieure)
- **Signification :** Pour toutes les grandes tailles (i.e., n >= n0), assurance que l'algorithme ne prend pas plus de c*g(n) étapes

Illustration



Exemple (i)

- Prouver que $f(n) = 5n + 37 =$ $O(n)$ i.e $g(n) = n$

- But : Trouver une constante c Є R et un seuil n_0 Є N à partir duquel $|f(n)| <= c|n|$

- On remarque que $5n + 37 <= 6n$ si $n >=37$

- On déduit que $c = 6$ fonctionne à partir de $n_0 = 37$

- Remarque : on ne demande pas d'optimiser (le plus petit c ou n_0 qui fonctionne) juste il faut donner des valeurs qui fonctionnent

Exemple (ii)

- Prouver que $f(n) = 6n^2 + 2n - 8$ est en $O(n^2)$
- But : Trouver une constante $c \in R$ et un seuil $n_0 \in N$ à partir duquel $| 6n_0^2 + 2n_0 - 8 | <= c|n_0^2|$
- Chercher d'abord c, on remarque que $c = 6$ ne marche pas alors que $c = 7$ c'est bon
- On doit trouver un seuil n_0 à partir duquel $| 6n_0^2 + 2n_0 - 8 | <= 7|n_0^2|$
- On peut rapidement vérifier que $| 6n^2 + 2n - 8 | <= 7|n^2| \ \forall \ n \geq n_0 = 1$
- Pour $n_0 = 1$ et $c = 7$, on a $| 6n_0^2 + 2n_0 - 8 | <= c|n_0^2|$ □ $f(n) = O(n^2)$

Exemple (iii)

- Prouver que $f(n) = c_1n^2 + c_2n$ est en $O(n^2)$

- On remarque que $c_1n^2 + c_2n <= c_1n^2 + c_2 n^2 = (c_1 + c_2) n^2$ pour tout $n >= 1$

- $f(n) <= cn^2$ avec $c = c_1 + c_2$ et $n_0 = 1$

- Donc $f(n) = O(n^2)$

Utilisation de la notation O

Réellement, pour l'approximation de la complexité, O(.) est utilisé pour donner une approximation de l'ordre de complexité sans le majorer

Exemples :

$$f(n) = n^3 + 2\,n^2 + 4\,n + 2 = O(n^3)$$

(si $n \geq 1$ alors $f(n) \leq 8 \times n^3$)

$$f(n) = n \log n + 12\,n + 888 = O(n \log n)$$

$$f(n) = 1000\,n^{10} - n^7 + 12\,n^4 + \frac{2^n}{1000} = O(2^n)$$

Notation Ω

- Soit f(n) et g(n) deux fonctions
- $f(n) = \Omega\ (g(n))$ ssi
  $\exists\ n_0 \in \mathbb{N}$ et $c > 0 \in \mathbb{R}$ tq $f(n) \geq c\ .\ g(n)\ \ \forall\ n \geq n0$

- Exemple
  - $f(n) = n^4 \quad g(n) = n^3$

  $g(n)=O(f(n)) \qquad f(n)\ =\Omega(g(n))$

- Exemples
  - $f(n) = n^4 + 15n^3 + 12 = \Omega\ (n^3)$
  - $f(n) = n^4 + 15n^3 + 12 = \Omega\ (n^4) = O(n^4)$

Notation Θ

- Soit f(n) et g(n) deux fonctions
- $f(n) = \Theta(g(n))$ ssi
  $\exists\ n_0 \in \mathbb{N}$ et $c1, c2 > 0 \in \mathbb{R}$ tel que
  $c1 \cdot g(n) \leq f(n) \leq c2 \cdot g(n)\ \forall\ n \geq n0$
- Exemple
  - $f(n) = 14n^7 = \Theta(n^7)$
  - $f(n) = 13n^5 + 12n^4 + 7 = \Theta(26n^5 - 8)$

Propriétés

- Transitivité
  - si $f(n) = O(g(n))$ et $g(n) = O(h(n))$ alors $f(n) = O(h(n))$
  - De même pour Ω et Θ
- Réflexivité
  - $f(n) = O(f(n)) = \Theta(f(n)) = \Omega(f(n))$
- Symétrie
  - si $f(n) = \Theta(g(n))$ alors $g(n) = \Theta(f(n))$
- Antisymétrie
  - si $f(n) = O(g(n))$ alors $g(n) = \Omega(f(n))$
  - si $f(n) = \Omega(g(n))$ alors $g(n) = O(f(n))$

Analogies

O : $\leq$

Ω : $\geq$

Θ : $=$

Notation de Landau et limites

- Soit f(n) et g(n) deux fonctions

- Si $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} 0 \text{ alors } f(n) = O(g(n)) \text{ et } g(n) \neq O(f(n)) \\ c \neq 0 \text{ alors } f(n) = \Theta(g(n)) \\ \infty \text{ alors } g(n) = O(f(n)) \text{ et } f(n) \neq O(g(n)) \end{cases}$

- Exemples : $\lim\limits_{n\to\infty} \frac{nlog\, n}{n} = \infty$ *donc* n=O(nlog n)

  $\lim\limits_{n\to\infty} \frac{n^3}{n^4} = 0$ *donc* $n^3 = O(n^4)$

  $\lim\limits_{n\to\infty} \frac{6n4}{n^4} = 6$ *donc* $4n^4 = \Theta(n^4)$

Règle de simplification

- La règle suivante permet de simplifier les complexités en omettant des termes dominés :
- Quand on somme les complexités des sous-algorithmes; on garde la complexité dominante
- Soit f(n) et g(n) deux fonctions :

$O(f(n) + g(n)) = O(f(n))$ si $g(n) = O(f(n))$

ou $O(g(n))$ si $f(n) = O(g(n))$

Exemple : $O(n + n \log n) = O(n \log n)$ car $n = O(n \log n)$

Autres simplifications

- Les quelques règles suivantes permettent de simplifier les complexités en omettant des termes dominés :
  - Les **coefficients peuvent être omis** : $O(14n^2)$ devient $O(n^2)$
  - **n^a domine n^b si $a > b$** : par exemple, n^2 domine n
  - **Les constantes additives peuvent être omises** : $O(4n^4 + 20)$ devient $O(4n^4)$
  - Une **exponentielle domine un polynôme** : 3^n domine n^5 (et domine également 2^n)
  - **un polynôme domine un logarithme** : n domine $(\log n)^c$ (c constante)

    => Cela signifie également, que n^2 domine nlog n

Exemples de simplification

- Soit $g(n) = 4n^4 - 5n^2 + 60n + 185$
- Omission des constantes multiplicatives :

  $1n^4 - 1n^2 + 1n + 185$
- Omission des constantes additives :

  $n^4 - n^2 + n + 0$
- Retenu du terme de plus haut degré : n^4

  $\Rightarrow g(n) = O(n^4)$

Règles de calcul

- Les termes constants :

$$O(c) = O(1)$$

- Les constantes multiplicatives sont omises:

$$O(c*H) = c * O(H) = O(H)$$

- L'addition est réalisée en prenant le maximum:

$$O(H1)+O(H2) = O(H1+H2) = \max(O(H1), O(H2))$$

- La multiplication reste inchangée:

$$O(H1) * O(H2) = O(H1*H2)$$

Classes de complexité
Complexité polynômiale/ exponentielle

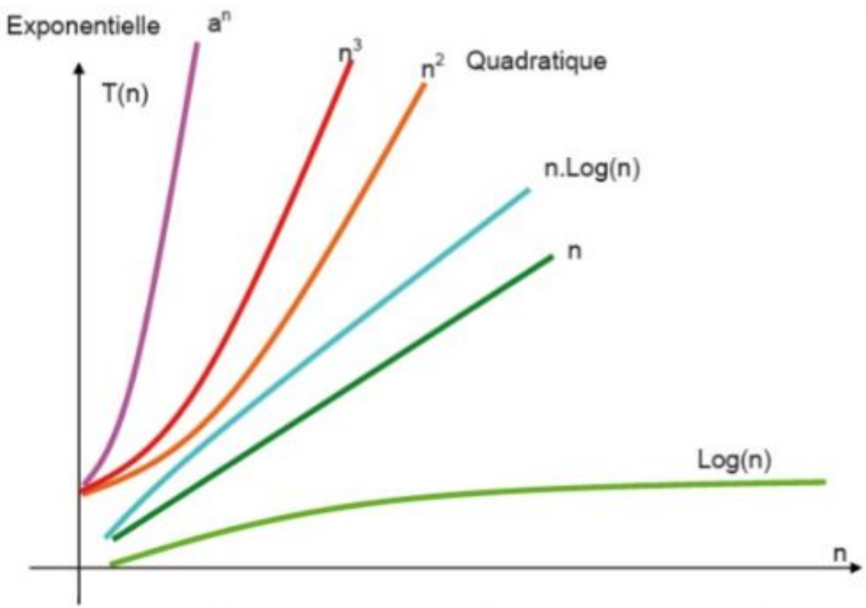
- Une complexité est dite **polynômiale** si elle s'écrit sous la forme d'un polynôme en les paramètres du problème.

Exemple : n, $7n^2$, n^4+5n^3+2n, n^{100}

- Si le polynôme est d'ordre q la complexité est dite polynômiale d'ordre q (q=1 : linéaire, q=2 quadratique…)
- O(n log n) est dite complexité est polylogarithmique

- Une complexité est dite **exponentielle** si les paramètres du problème apparaissent en exposant (puissance) dans l'expression de complexité

Exemples : $O(2^n)$, $O(3^n)$,…, $O(n^n)$



Classes de complexité

- Les fonctions équivalentes peuvent être rangées dans la même classe
- Deux algorithmes de la même classe sont considérés de même complexité
- Les classes de complexité les plus fréquentes (par ordre croissant selon O(.))

Complexité	Classe
$O(1)$	**Constante**
$O(\log n)$	**Logarithmique**
$O(n)$	**Linéaire**
$O(n\log n)$	**Sous-quadratique**
$O(n^2)$	**Quadratique**
$O(n^3)$	**Cubique**
$O(2^n)$	**Exponentielle**
$O(n!)$	**Factorielle**

Ordre de grandeur

Résultats obtenus pour une machine à 10^9 opérations flottantes par secondes (flops)

T.\C.	$\log n$	n	$n \log n$	n^2	2^n
10	$3\mu s$	$10\mu s$	$30\mu s$	$100\mu s$	$1000\mu s$
100	$7\mu s$	$100\mu s$	$700\mu s$	$1/100s$	10^{14} *siècles*
1000	$10\mu s$	$1000\mu s$	$1/100s$	$1s$	*astronomique*
10000	$13\mu s$	$1/100s$	$1/7s$	$1,7mn$	*astronomique*
100000	$17\mu s$	$1/10s$	$2s$	$2,8h$	*astronomique*

Optimalité/ Coût minimum

- Le coût d'un algorithme représente sa complexité

- Les notions d'optimalité et de coût minimum

sont deux notions importantes et différentes mais qui représentent parfois une confusion

- Soit $\mathcal{A}$(n) un algorithme résolvant $\mathcal{P}$(n)

Dire que $\mathcal{A}$(n) est optimal et dire qu'il est de coût minimum ne veut pas du tout dire la même chose.

On définit chacune de ces notions dans ce sui suit.

Algorithme optimal

- Soit $\mathcal{P}$(n) un problème impliquant n paramètres
- Soient $\mathcal{A}_1$(n), $\mathcal{A}_2$(n), …….. , $\mathcal{A}_t$(n) les t uniques algorithmes résolvant $\mathcal{P}$(n)
- Les complexités respectives de ces algorithmes sont respectivement $\mathcal{C}_1$(n), $\mathcal{C}_2$(n), …….. , $\mathcal{C}_t$(n)
- Un algorithme $\mathcal{A}_k$(n) ($1 \leq k \leq t$) est dit **optimal** ssi $\mathcal{C}_k(n) \leq \mathcal{C}_i(n), \forall\ 1 \leq i \leq t$

Sa complexité est inférieure a toutes celles des algos résolvant le même problème

Exemple : Le tri par insertion n'est pas optimal $O(n^2)$. Le tri rapide est optimal $O(n \log n)$

Algorithme de coût minimum

- Soit $\mathscr{A}$(n) un algorithme résolvant le problème $\mathscr{P}$(n) de complexité $\mathscr{C}$(n)
- Soit E(n) l'ensemble des entrées (données) de $\mathscr{A}$(n)
- $\mathscr{A}$(n) est dit de coût minimum ssi

$$\mathscr{C}(n) = \Theta(|E(n)|)$$

Sa complexité a le même ordre que le nombre de ses entrées

Exemples

- A1 : Somme des éléments d'une liste de taille n
  $\mathcal{C}(n) = n\text{-}1$, $|E(n)| = n$
  => A1 est de coût minimal vu que $\mathcal{C}(n) = \Theta(|E(n)|)$
- A2 : Tri par insertion d'une liste de taille n
  $\mathcal{C}(n) = \Theta(n^2)$, $|E(n)| = n$
  => A2 n'est pas de coût minimal vu que $\mathcal{C}(n) \neq \Theta(|E(n)|)$
- A3 : Somme de 2 matrices carrée (n, n)
  $\mathcal{C}(n) = n^2$, $|E(n)| = 2n^2$
  => A3 est de coût minimal
- A4 : Produit de 2 matrices carrée (n, n)
  $\mathcal{C}(n) = \Theta(n^3)$, $|E(n)| = 2n^2$
  => A2 n'est pas de coût minimal