U.F.R SCIENCES ET TECHNIQUES
Département d'Informatique
B.P. 1155
64013 PAU CEDEX
Téléphone secrétariat : 05.59.40.79.64
Télécopie : 05.59.40.76.54

V-Complexité des algorithmes et Classes de complexité : partie 2

I-Notion de complexité d'un algorithme
II-Complexité en temps d'un algorithme
III-Calcul de la complexité
IV-Classes de complexité

IV- Classes de complexité

Ici l'objet de l'étude :

-ne sont pas les algorithmes

-mais les **problèmes** que ces algorithmes sont censés résoudre.

On parle alors de **problèmes algorithmiques**

Il s'agit de comprendre comment les **problèmes algorithmiques** se placent les uns par rapport aux autres.

Dans ce objectif, la **théorie de la complexité** établit des **hiérarchies** de difficultés.

Les niveaux de ces hiérarchies sont appelés **classes de complexité.**

La théorie de la complexité propose une définition des **classes de complexité** .

Elle permet de :

- de **classer** les problèmes
- en fonction de la **complexité des algorithmes** qui existent pour les résoudre.

Cette classification distingue :

- les algorithmes **déterministes**,
- des algorithmes **non-déterministes**.

Un algorithme est dit **déterministe** lorsque le résultat :

- est prévu
- et dépend uniquement des **données d'entrées**,

Un algorithme est dit **non-déterministes** lorsque le résultat peut dépendre aussi de:

- l'état de **variables globales**,
- de choix arbitraires,
- parallélisme des actions: asynchronisme
- etc

A chaque étape de son déroulement, un tel algorithme :

- peut effectuer un **choix non-déterministe**
- entre plusieurs actions possibles.

On entend peu parler d'algorithme **non déterministe** dans les cours de programmation.

Pourquoi ?:

Parce que la question suivante:

« **Que calcule un algorithme non déterministe ?** »

n'a guère reçu de réponse satisfaisante en général.

Influence de la complexité des algorithmes

Pour **illustrer** l'importance de la mesure du **coût de la complexité**, considérons le temps correspondant à la complexité des algorithmes en:

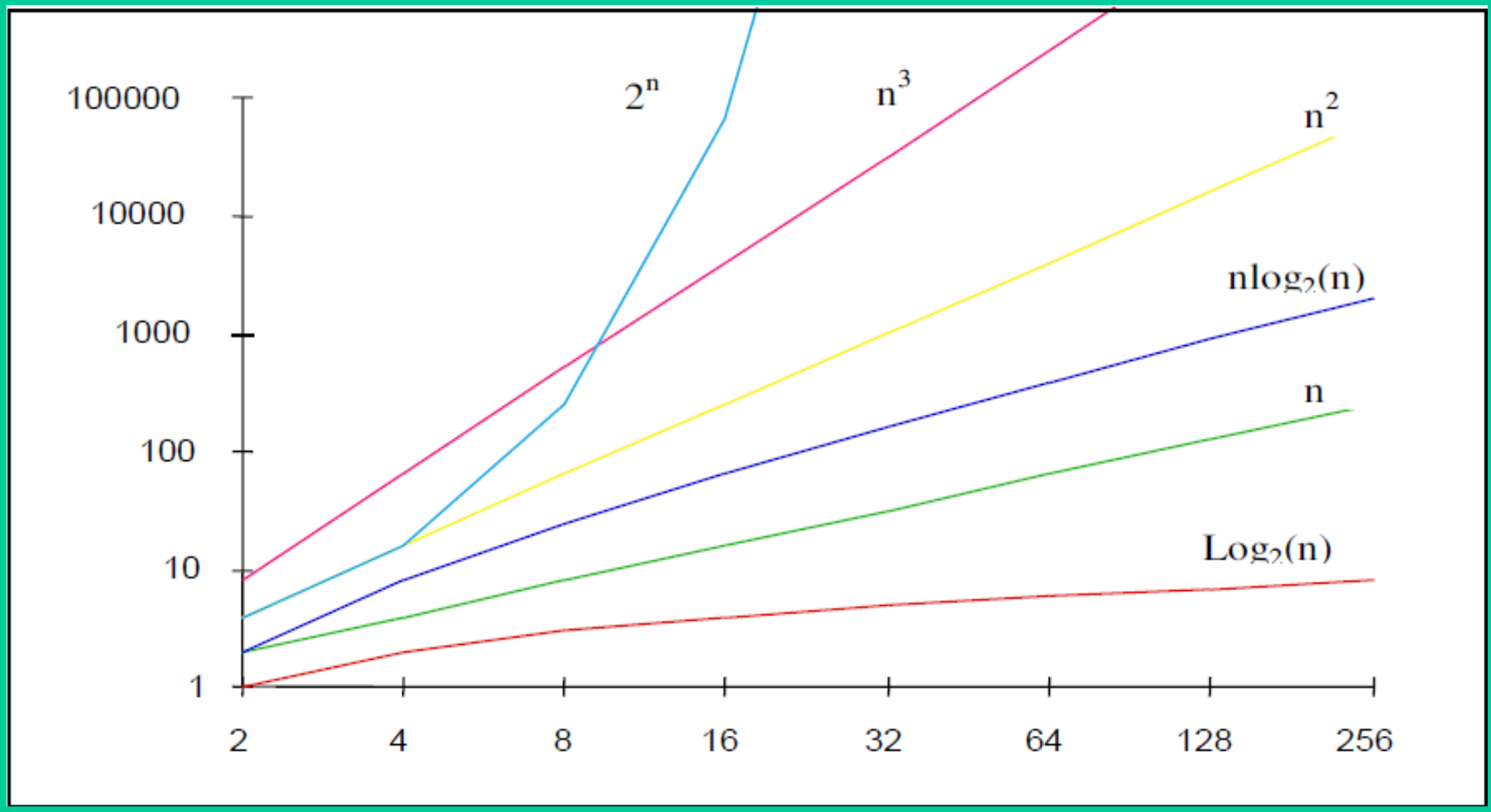
- O(n), O(n log2 n), $O(n^2)$,…, $O(2^n)$, O(n!)
- pour des entrées de taille **n** croissante.

Le processeur utilisé a une de puissance 1 Mips

Le tableau ci-dessous est un extrait de [Kleinberg and Tardos].

Complexité	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	4 s
$n = 30$	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	18 min	10^{25} ans
$n = 50$	< 1 s	< 1 s	< 1 s	< 1 s	11 min	36 ans	∞
$n = 100$	< 1 s	< 1 s	< 1 s	1s	12, 9 ans	10^{17} ans	∞
$n = 1000$	< 1 s	< 1 s	1s	18 min	∞	∞	∞
$n = 10000$	< 1 s	< 1 s	2 min	12 jours	∞	∞	∞
$n = 100000$	< 1 s	2 s	3 heures	32 ans	∞	∞	∞
$n = 1000000$	1s	20s	12 jours	31, 710 ans	∞	∞	∞

Les courbes ci-dessous illustrent mieux la croissance du coût en temps en fonction de taille **n** des données



Conclusion:

Il appert qu'un algorithme de complexité **exponentielle** est très rapidement **inutilisable**.

Donc, ce n'est pas un algorithme **très raisonnable**.

Ce sont les algorithmes de complexité **polynomiale** qui suscitent le **plus d'intérêt**.

Notion de complexité de problème

La **complexité d'un problème** est une notion qui permet de discuter :

-de l'**optimalité**
-ou la **non-optimalité**

d'un **algorithme** pour résoudre un problème donné.

On fixe un problème p: par exemple celui de «trier une liste d'entiers».

Un algorithme $\mathcal{A}$ qui résout p

Pour chaque donnée **d**, l'algorithme $\mathcal{A}$ produit la réponse **correcte**, notée $\mathcal{A}$ **(d).**

La complexité du problème sur les entrées de taille **n** est définie par:

$$\mu(\mathcal{P},n) = \inf_{\mathcal{A}\text{ algorithme qui résout }\mathcal{P}} \max_{d\text{ entrée avec }taille(d)=n} \mu(\mathcal{A},d)$$

Autrement dit, on ne fait plus :

-seulement varier les entrées **d** de taille n,

-mais varier aussi l'algorithme $\mathcal{A}$.

On considère que le **meilleur** algorithme qui résout un problème est celui avec la meilleure complexité **au pire cas**.

complexité du problème =
complexité du meilleur algorithme au pire cas.

Notion de temps raisonnable

La **théorie de complexité** permet de comprendre ce que l'on appelle en informatique un **algorithme raisonnable**,

Elle introduit la **NP-complétude** qui permet de discuter, en **théorie**, de la frontière entre :

- le **raisonnable**
- et le **non raisonnable**.

Convention

Pour différentes raisons, la **convention** suivante s'est imposée en informatique :

Un algorithme est **efficace** si sa complexité en temps T(n) est **polynomiale**:

$$T(n) = O(n^k)$$

pour **k** entier.

En pratique, on peut argumenter que par exemple un algorithme de complexité polynomiale :

$$T(n) = O(n^{1794})$$

n'est **pas très raisonnable**.

Mais, au sens de la **théorie de la complexité**, on considère qu'il est **raisonnable** .

Question:

Pourquoi ne pas prendre, par convention, un coût en temps **linéaire**:

$$T(n) = O(n)$$

ou **quadratique**:

$$T(n) = O(n^2)$$

comme notion de coût "**raisonnable**" ?

Réponse:

Deux raisons profondes ont conduit à cette convention.

Première raison

La première raison profonde s'appuie sur le fait qu'un «algorithme polynomial s'affranchit du codage»

Deuxième raison

La deuxième raison profonde permet de «s’affranchir du modèle de calcul »

1-Classe P

P est exactement la **classe des problèmes** qui admettent pour solution un **algorithme polynomial**.

La classe **P** est caractérisée par un coût en temps:

$$T(n) = O(n^k)$$
$$k \in \mathbb{N}$$

Classe NP

Il y a une classe de problèmes pour lesquels, **à ce jour :**

-on n'arrive pas, encore, à construire d'algorithme **polynomial**.

-sans qu'on arrive à **prouver** que cela ne soit pas possible.

C’est historiquement ce qui a mené à considérer la classe de problèmes que l’on appelle **NP**: **Non-déterministe Polynomial**

Parmi les problèmes les plus connus dans la classe NP, citons deux:

1- problème de **K-Coloriabilité**

2- problème de **Cycle hamiltonien**

La classe **NP** réunit les problèmes de décision pour lesquels la réponse **oui** peut être décidée par un algorithme :

- **non-déterministe**
- en un temps **polynomial**.

De façon **équivalente,** c'est la classe des problèmes qui :

- étant donné une **solution** du problème **NP**: un **certificat**
- admettent un **algorithme polynomial** capable de répondre **oui** ou **non**.

Classe Co-NP

Classe « Complémentaire de NP » est l'équivalente de la classe NP, mais avec la réponse **non.**

Classe EXPTIME

Elle rassemble les problèmes admettant un algorithme **déterministe** dont le coût en temps est **exponentiel**.

C-Complétude et réduction

Soit **C** une classe de complexité (comme **P**, **NP**, etc.).

Un problème est **C-difficile** s'il est au moins aussi dur que tous les problèmes dans **C**.

On dit qu'un problème est **C-complet** si

-il est dans **C**, et

-il est **C-difficile.**

Réduction

Soient P1 et P2 deux problèmes ;

Formellement, une **réduction** de P2 à P1 est un **algorithme** transformant :

- toute instance de P2
- en une instance de P1.

Ainsi, si l'on a un **algorithme** pour résoudre P1, on sait aussi résoudre P2.

P1 est donc au moins aussi difficile à résoudre que P2.

P est alors **C-difficile** si pour tout problème P' de C, P' se réduit à P.

La **réduction** **la plus simple** consiste simplement à :

- transformer le problème à classer
- en un problème déjà classé.

Propriété de la réduction

L'idée est que si A se **réduit** à B, alors :

- le problème A est plus **facile** que le problème B,
- le problème B est plus **difficile** que le problème A.

Le problème A est donc plus **facile** que le problème B, est noté :

$$A \leq B.$$

Problème ouvert P = NP ?

La recherche travaille activement à déterminer :

-si **NP⊆ P** (concluant à **P = NP**)

-ou si, au contraire **P≠ NP**

On a trivialement :

P⊆ NP

car un algorithme **déterministe** est un algorithme non déterministe particulier.

En revanche la réciproque, que l'on résume par :

P = NP

est l'un des problèmes ouverts les plus fondamentaux et intéressants en **informatique théorique**.

Cette question a été posée en **1970** pour la première fois.

Celui qui arrivera à décider si :

- **P = NP**)

-ou, au contraire **P≠ NP** ?

recevra le prix Clay (plus de 1 000 000 $).

Le **problème P = NP** revient à savoir si :

- on peut résoudre un problème NP-Complet
- avec un algorithme polynomial.

Faire tomber **un seul** des problèmes NP-Complet dans la classe P fait tomber **l'ensemble de la classe NP**.