CHAPITRE 2
COMPLEXITE DES ALGORITHMES / DES PROBLEMES

Plan

1. Introduction ;
2. Définition ;
3. Calcul de la complexité ;
4. Notation de landau ;
5. Classes de problèmes ;

1. Introduction

- L'ordinateur dispose de deux ressources nécessaires pour exécuter un programme (d'un algorithme) : mémoire et processeur ;
- La complexité d'un algorithme est une mesure de la quantité de ces ressources consommée par l'exécution de cet algorithme ;
- On distingue alors :

➢ La complexité spatiale : espace mémoire ;

➢ La complexité temporelle : temps CPU ;

requis pour l'exécution de cet algorithme.

- La première n'importe plus avec l'évolution spectaculaire du hardware ;
- La seconde est dite alors ***complexité*** tout court ;
- Il est clair que le temps d'exécution d'un algorithme est fonction de ses opérations (plus particulièrement ses opérations fondamentales) et de la taille de son input (la quantité de données que prend l'algorithme en entrée) ;

2. Définition

- La complexité d'un algorithme est le nombre d'opérations fondamentales que fait cet algorithme en fonction de la taille de son input dans les pires cas ;
- La complexité est donc une estimation formelle du temps d'exécution en fonction du input size ;
- Elle est indépendante de :
  - De la machine …;
  - De l'OS …;
  - Du langage …;

… à utiliser pour exécuter l'algorithme.

2. Définition

Remarque

On distingue, la complexité :

- Dans les pires cas ;
- Dans les meilleurs cas ;
- Dans le cas moyen ;

Une définition plus formelle :

La complexité est le nombre d'opérations élémentaires qu'effectue une machine de Turing pour reconnaitre un mot en fonction de la longueur de ce mot.

Elle sert à analyser , évaluer , comparer les algorithmes.

Ce qui importe c'est quand la taille du input est assez grande : allure asymptotique du nombre d'Operations.

Exemple 1 ……

3. Calcul de la complexité

Soit P1 et P2 deux programmes et T le nombre d'opérations fondamentales Δ :

T(x=a / b / c / d) = 3

T(if (condition) P1 ;) = T(P1)

T(if (condition) P1 else P2 ;) = max (T(P1) , T(P2)

T(for (i=0 ; i<k ; i++) P1 ;) = k*T(P1)

T(while (condition) P1 ;) = maxiterations*T(P1)

T(P1; P2)= T(P1) + T(P2)

T(P1 // P2)= max (T(P1) , T(P2)

3. Calcul de la complexité

Si le programme est itératif et n est la taille du input :

T(n) = une fonction de n ;

Exemples : $T(n) = 3n+2$; $(Tn)= n^2+n-1$; $T(n)= 2^n$

Si le programme est récursif et n est la taille du input :

T(n) est récurrente = fonction de T(n-1) ou T(n/2) ou T(n-2), ….;

avec un cas trivial comme T(0)= 1 ou T(2)= 0 , ….

Exemples :

$T(1) = 1$ et $T(n)= T(n-1) + 2 \Rightarrow T(n) = 2n+1$;

$T(1) = 0$ et $T(n)= T(n-1) + n \Rightarrow T(n) = n(n-1) = n^2-n$;

$T(1) = 1$ et $T(n)= 2 *T(n/2) + 1 \Rightarrow T(n) = \log_2 n$;

4. Notation de Landau

Landau s'est intéressé à l'étude asymptotique des fonctions numériques à variable entière :

$f = O(n^2) \Leftrightarrow \exists n_0 \in N , \exists c \in R , \forall n \geq n_0 : T(n) \leq c.n^2$

$f = \Omega(g) \Leftrightarrow \exists n_0 \in N , \exists c \in R , \forall n \geq n_0 : f(n) \geq c.g(n).$

$f = \theta(g) \Leftrightarrow \exists n_0 \in N , \exists c1,c2 \in R , \forall n \geq n_0 : c1.g(n) \leq f(n) \leq c2.g(n).$

$3n^2+2$ $4n^2-3n+2$ $5n^2+100$ $8n^2-100000000$

$2n \leq 3n+2 \leq 4n$ $\forall n \geq 2$ $n0=2$ $c=4$ $T(n) \in O(n)$

T(n)= le nombre d'opérations en fonction la taille des données

$= 4n^2-3n+2 \geq c.n^2$ $\Omega(n^2)$

4. Notation de Landau

La notation la plus utilisée est O (se lit big o).

On écrit $f = O(n)$ ou $f \in O(g)$

Exemples :

Si $T(n)= 2n^2+3n+1$ alors $T(n) \in O(n^2)$ on dit alors que l'algorithme est en $O(n^2)$; $2n^2+3n+1 \leq c.n^2$

Ceci veut dire : Si n est assez grand, T(n) est de l'ordre de n^2 ;

Ces notations permettent de décrire l'allure la courbe représentative de la fonction f quand n est assez grand ($n \to \infty$)

Remarques

$f= O(g) \Leftrightarrow g= \Omega(f)$

$f= \theta(g) \Leftrightarrow f= O(g)$ and $g= O(f)$

Θ est symétrique : $f= \theta(g) \Leftrightarrow g= \theta(f)$

O , Ω sont antisymétriques.

O , θ , Ω sont réflexives et transitives .

Comparer les algorithmes de tri *n* nombres

a, b, c : constantes

- **Enumération** (idiot)

nombre d'étapes borné par $c.n!$

- **Sélection** (bête)

nombre d'étapes borné par $a.n.(n\text{-}1)/2 = O(n^2)$

- **Fusion** (malin)

nombre d'étapes borné par $b.n.\log n = O(n\log n)$

Un tout petit peu de combinatoire

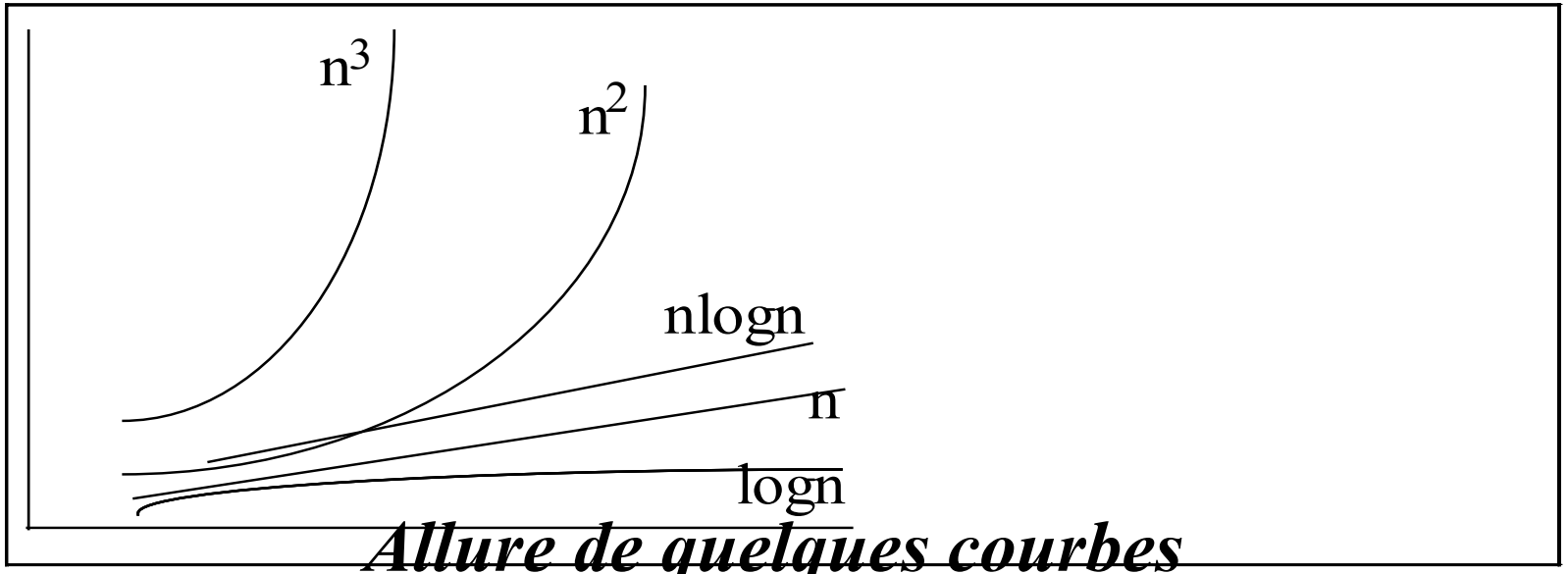
- **1 étape se déroule en 10^{-6} seconde**
- **Le nombre d'étapes est de l'ordre de:**

	Log(n)	n	n^2	2^n	n!

polynomial efficace

non polynomial non efficace

"Bonne complexité" O(log n) ou O(n) ou O(n log n)



Allure de quelques courbes

n = 10^6 **1 μs par opérations**

$\log_2 n$	n	$n\log_2 n$	n^2	n^3
20μs	1 s	20 s	12 j	32 Ka

Complexités usuelles

T(n)	Notation en O	Complexité
Constante (1, 10, 500,..)	O(1)	constante
an+b	O(n)	Linéaire
aLog n+b	O(logn)	Logarithmique
an^2+bn+c	$O(n^2)$	Quadratique
anlogn+b	O(nlogn)	Quasi logarithmique
Polynôme en n degré k	$O(n^k)$	polynomiale
$c.a^n$ (a constante >1)	$O(a^n) \approx O(2^n)$	Exponentielle
c.n !	$O(n !) \approx O(2^n)$	Exponentielle

10^9 Instructions/seconde (1 gigaHertz)

n	5	10	15	20	100	1000
log n	**$3\ 10^{-9}$ s**	**$4\ 10^{-9}$ s**	**$4\ 10^{-9}$ s**	**$5\ 10^{-9}$ s**	**$7\ 10^{-9}$ s**	**10^{-8} s**
2n	**$10\ 10^{-9}$ s**	**$2\ 10^{-8}$s**	**$3\ 10^{-8}$ s**	**$4\ 10^{-8}$ s**	**$2\ 10^{-7}$ s**	**$2\ 10^{-6}$ s**
nlogn	**$12\ 10^{-9}$ s**	**$3\ 10^{-8}$s**	**$6\ 10^{-8}$s**	**10^{-7}s**	**$7\ 10^{-7}$s**	**10^{-5}s**
n^2	**$25\ 10^{-9}$ s**	**10^{-7} s**	**$2{,}25\ 10^{-7}$ s**	**$4\ 10^{-7}$ s**	**10^{-5} s**	**10^{-3} s**
n^5	**$3\ 10^{-6}$ s**	**10^{-4} s**	**$7{,}59\ 10^{-4}$ s**	**$3\ 10^{-3}$ s**	**10 s**	**10^6 s = 11 jours**
2^n	**$32\ 10^{-9}$ s**	**10^{-6} s**	**$3{,}28\ 10^{-5}$ s**	**10^{-3} s**	**$1{,}2\ 10^{21}$ s $4\ 10^{11}$ siècles**	**10292 s = $3\ 10^{282}$ siècles**
n !	**$120\ 10^{-9}$ s**	**$4\ 10^{-3}$ s**	**$1{,}4\ 10^3$s= 23 minutes**	**$2{,}4\ 10^9$ s = 77 ans**	**10147 s $3\ 10^{139}$ siècles**	**10^{500} s**
n^n	**$3\ 10^{-6}$ s**	**10 s**	**$4{,}37\ 10^8$s = 13 ans**	**10^{17}s = $3\ 10^7$siècles**	**10191s $3\ 10^{181}$siècles**	**10^{3000} s**

5. Classes de problèmes

Problème de décision

Un problème de décision est un problème dont la solution est vrai ou faux.

Exemples :

- L'élément x est-il dans le tableau A[] ?
- Le tableau A[] est-il tiré par ordre croissant ?
- x est-il le max dans le tableau A[] ?
- n est-il premier ? Le problème de primalité Prime PB
- Peut-on colorer les sommets d'un graphe par 3 couleurs tels que deux sommets adjacents aient 2 couleurs différentes?
- Existe-il un cycle hamiltonien de longueur minimale dans le graphe G ?
- Peut-on placer 8 reines sur un échiquier sans qu'elles se menacent ?

La complexité d'un problème est celle du meilleur algorithme le résolvant.

Meilleur = le plus rapide.

Classification des problèmes de décision

Classes P , NP , NP-complet , NP-difficile.

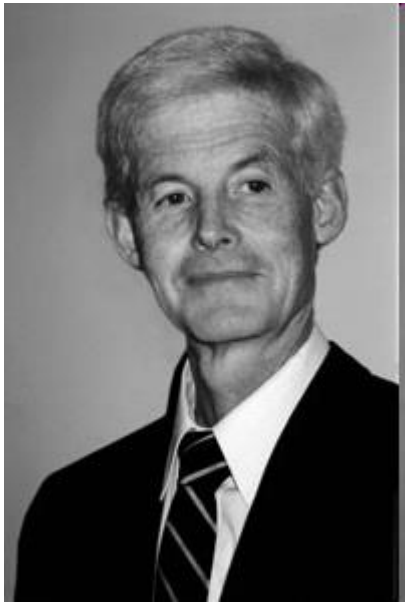
On dit qu'un problème de décision est dans la classe P (Polynomial time) ou « facile easy » si et seulement s'il existe un algorithme polynomial déterministe le résolvant.

Exemples :

- L'élément x est-il dans le tableau A[] ?
- Le tableau A[] est-il tiré par ordre croissant ?
- Existe-il un cycle eulérien de longueur minimale dans le graphe G ?

Approche de la théorie de la complexité

Théorie développée à la fin du 20ème siècle (S. Cook 1970 et L. Levin 1973)



J. Edmonds, R Karp, …

On dit qu'un problème est dans la classe NP (Non deterministic Polynomial time) si et seulement s'il existe un algorithme polynomial non déterministe le résolvant , c.-à-d., connaissant une solution, elle peut être vérifiée par un algorithme polynomial déterministe.

NP

Le seul algorithme déterministe pouvant résoudre ce problème est exponentiel.

Exemples :

- Peut-on colorer les sommets d'un graphe par 3 couleurs tels que deux sommets adjacents aient 2 couleurs différentes?
- Existe-il un cycle hamiltonien de longueur minimale dans le graphe G ?
- Peut-on placer 8 reines sur un échiquier sans qu'elles se

Réduction polynomiale

On que le problème A est une réduction polynomiale du problème B (ou A se réduit polynomialement à B) si et seulement si A peut être résolu en un temps polynomial en fonction du nombre d'appels de B et on écrit : $A \leq_P B$.

Problème NP-difficile

Un problème X est dit NP-difficile si et seulement si tout problème de NP peut être réduit polynomialement à X.

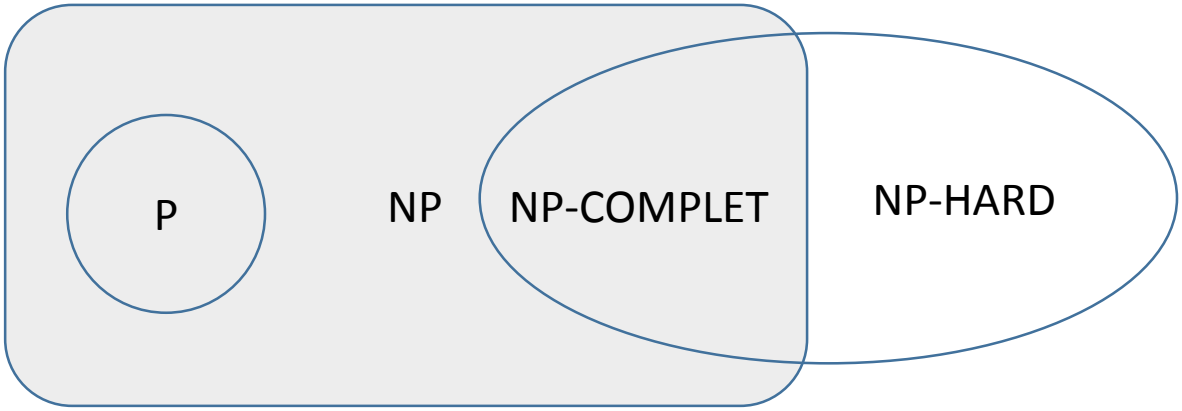
$A \in$ NP-H ssi $\forall$ B $\in$ NP $A \leq_P B$

Et si $A \in$ NP alors $A \in$ NP-C

c.-à-d. (NP-H) $\cap$ NP = NP-C et NP-C $\subset$ NP

Il est clair que P $\subset$ NP mais NP $\subset$ P ?

P=NP ? Problème ouvert millénaire 1.000.000 dollars !



Pour montrer qu'un problème Π est polynomial

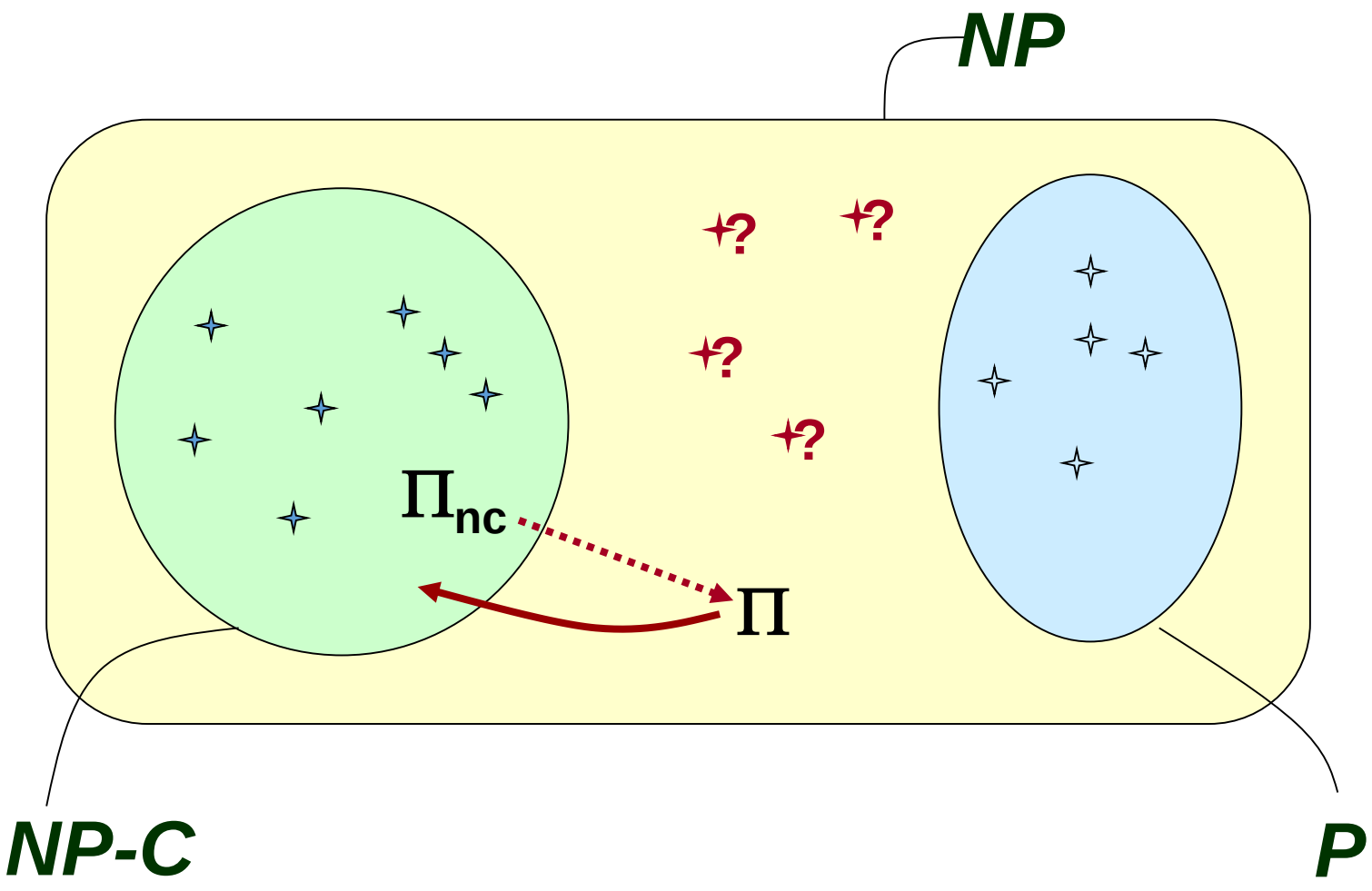
il faut trouver un algorithme pour le résoudre et prouver que cet algorithme s'exécute en un temps qui augmente de façon polynomiale en fonction de la taille de l'instance traitée

Pour montrer qu'un problème Π est *NP*-complet, on choisit un problème déjà connu pour être *NP*-complet, soit Π_{nc} , et on montre que Π_{nc} peut se "transformer" en Π.

Donc, si on savait résoudre Π, on saurait résoudre Π_{nc}.

Or, on ne sait pas résoudre Π_{nc} : donc il va sûrement être difficile de résoudre Π.

Π va, à son tour, être classé NP-complet.



Si on savait résoudre facilement Π on saurait résoudre aussi Π_{nc}; or on ne sait pas résoudre Π_{nc}

Π est donc sûrement difficile à résoudre

Le problème SAT

"satisfiabilité" d'une expression logique

Exemple

$(xV\bar{y}Vz) \wedge (\bar{x}V\bar{y}Vt) \wedge (yV\bar{z}Vt) \wedge (xVzV\bar{t})$

x est vrai ou faux **x vrai $\Leftrightarrow$ $\bar{x}$ faux**

Peut-on affecter des valeurs vrai ou faux aux variables de telle façon que l'expression soit vraie ?

Exemple

une solution: x=vrai y=faux t=vrai z=vrai

Le théorème de Cook

Stephen Cook
a classé le
problème SAT
comme *NP*-complet

SAT est le premier problème
***NP*-complet connu**