IN 101 - Cours 04

2 octobre 2009

présenté par
Matthieu Finiasz

La notion de complexité

IN101 - 2009-2010

Qu'est-ce que la complexité ?
La notion centrale en algorithmique

- La complexité d'un algorithme mesure son efficacité intrinsèque :
  - ▷ en fonction de la taille des données à traiter,
  - ▷ asymptotiquement,
  - ▷ dans le pire cas, ou en moyenne.
- En pratique, pour une entrée de taille n :
  - ▷ on compte le nombre d'opérations de base nécessaires,
  - ▷ on regarde comment ce nombre évolue asymptotiquement.
- Il s'agit en général de complexité temporelle
  - ▷ on s'intéresse souvent aussi à la complexité spatiale.

IN101 - 2009-2010

Un premier exemple simple

```
1 unsigned int sum_of_squares(unsigned int n) {
2   unsigned int sum = 0;
3   for (int i=1; i<n+1; i++) {
4     sum += i*i;
5   }
6   return sum;
7 }
```

- Pour une simple boucle `for` la complexité est facile à calculer :
  - ▷ la taille de l'entrée est n
  - ▷ la fonction fait n multiplications et n additions
    - ⇾ la complexité est linéaire en la taille de l'entrée.
- Un peu de réflexion permet de calculer cela en temps constant :
  - ▷ $\texttt{sum\_of\_squares}(n) = \frac{n(n+1)(2n+1)}{6}$
    - ⇾ 2 additions, 3 multiplications, une division.

Notations de complexité
Comparaison asymptotique

- Définitions :
  - $f(n) = O\left(g(n)\right)$ si
    $\exists c > 0,\ \exists n_0 \in \mathbb{N}^*,\ \forall n \geq n_0, \quad 0 \leq f(n) \leq c \times g(n)$.
  - $f(n) = \Theta\left(g(n)\right)$ si
    $\exists c, c' > 0,\ \exists n_0 \in \mathbb{N},\ \forall n \geq n_0, \quad c \times g(n) \leq f(n) \leq c' \times g(n)$.

- Exemples :
  - $n^2 + n + 1 = \Theta\left(n^2\right) = \Theta\left(2000n^2 + 1000000\right)$
  - $\log(n) = O\left(n\right)$
  - $\frac{n}{\log(n)} = O\left(n\right)$

$$\log(n) \ll \sqrt{n} \ll n \ll n\log(n) \ll n^2 \ll n^3 \ll 2^n \ll \exp(n) \ll n! \ll n^n \ll 2^{2^n}$$

IN101 - 2009-2010

Notations de complexité
Comparaison asymptotique

- Définitions :
  - Complexité polynomiale ⟶ souvent réalisable
    $\exists k > 0, \quad f(n) = O(n^k)$.
  - Complexité exponentielle ⟶ en général irréalisable
    $\exists b > 1, \quad b^n = O(f(n))$.
  - Complexité doublement exponentielle
    par exemple : $f(n) = 2^{2^n}$.
  - Complexité sub-exponentielle
    par exemple : $f(n) = 2^{\sqrt{n}}$.

$$\log(n) \ll \sqrt{n} \ll n \ll n\log(n) \ll n^2 \ll n^3 \ll 2^n \ll \exp(n) \ll n! \ll n^n \ll 2^{2^n}$$

IN101 - 2009-2010

Quelques ordres de grandeur

- Un PC standard fait de l'ordre de 2^{30} opérations binaires par seconde
  - 2^{40} opérations simples ⟶ réalisable par tout le monde.

- Les records actuels sont un peu au-delà de 2^{60} opérations binaires
  - réalisable par des gens « motivés » ⟶ NSA, Folding@home...

- En cryptographie on considère que 2^{80} opérations binaires sont irréalisables (aujourd'hui)
  - Clefs de 128 bits ⟶ sûres pour quelques dizaines d'années.

Premiers exemples :

Un problème - quatre algorithmes

IN101 - 2009-2010

Le problème
La suite de Fibonacci

- On cherche à calculer F_n, le n-ième nombre de la suite de Fibonacci :
  - $F_0 = 0$ et $F_1 = 1$
  - $F_n = F_{n-1} + F_{n-2}$ pour $n > 1$
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,...

- On sait faire le calcul direct :
  - suite récurrente double à coefficients constants
  - on résout $x^2 = x + 1$
    → $r_1 = \varphi = \frac{1}{2}(1 + \sqrt{5})$ et $r_2 = 1 - \varphi = \frac{1}{2}(1 - \sqrt{5})$
  - $F_n = \frac{1}{\sqrt{5}}(\varphi^n - (1 - \varphi)^n)$.

- $r_2 \simeq -0.62$ donc pour $n > 1$, F_n est l'entier le plus proche de $\frac{\varphi^n}{\sqrt{5}}$.

Fibonacci – Algorithme 1
Version récursive bête

```
1 unsigned int fibo1(unsigned int n) {
2   if (n < 2) {
3     return n;
4   }
5   return fibo1(n-1) + fibo1(n-2);
6 }
```

- Ici, la complexité est le nombre d'appels à `fibo1`
  - le résultat se décompose en une somme de F_1 et F_0 :

$$\begin{aligned} F_4 &= F_3 + F_2 \\ &= (F_2 + F_1) + (F_1 + F_0) \\ &= (F_1 + F_0) + F_1 + F_1 + F_0 \end{aligned}$$

- $F_1 = 1$ et $F_0 = 0$ donc il y a au moins F_n appels récursifs
  → la complexité est $\Theta(F_n) = \Theta(\varphi^n)$.

Fibonacci – Algorithme 2
Version avec mémoire

- On ne veut pas recalculer plusieurs fois la même valeur F_i
  - on utilise un tableau pour stocker tous les F_i.

```
1  unsigned int fibo2(unsigned int n) {
2    unsigned int* fib = (int*) malloc((n+1) * sizeof(int));
3    fib[0] = 0;
4    fib[1] = 1;
5    for (int i=2; i<n+1; i++) {
6      fib[i] = fib[i-1] + fib[i-2];
7    }
8    int res = fib[n];
9    free(fib);
10   return res;
11 }
```

- Complexité de $\Theta(n)$
  - mais complexité spatiale de $\Theta(n)$ aussi.

Fibonacci – Algorithme 3
Version sans mémoire

- On constate que chaque élément du tableau n'est lu que 2 fois :
  - aux étapes $i + 1$ et $i + 2$ de la boucle.

```
1  unsigned int fibo3(unsigned int n) {
2    unsigned int fib0 = 0;
3    unsigned int fib1 = 1;
4    int i;
5    for (i=2; i<n+1; i++) {
6      fib1 = fib0 + fib1;
7      fib0 = fib1 - fib0;
8    }
9    return fib1;
10 }
```

- Complexité de $\Theta(n)$ toujours
  - mais complexité spatiale de $\Theta(1)$ maintenant.

Fibonacci – Algorithme 4
Version optimale

- On utilise le calcul direct, mais on évite les nombres flottants.
- On a :

$$\begin{aligned} F_n &= 1 \times F_{n-1} + 1 \times F_{n-2} \\ F_{n-1} &= 1 \times F_{n-1} + 0 \times F_{n-2} \end{aligned}$$

Donc :

$$\begin{aligned} \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \times \begin{pmatrix} F_{n-1} \\ F_{n-2} \end{pmatrix} \\ &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \times \begin{pmatrix} F_1 \\ F_0 \end{pmatrix} \end{aligned}$$

- Simple problème d'exponentiation matricielle (résolu en TD 04)
  - ▷ complexité $\Theta(\log(n))$
  - ▷ complexité spatiale $\Theta(1)$.

Note : les valeurs propres de la matrice sont φ et $1 - \varphi$.

Comparaison des quatre algorithmes
Théorie et pratique sont d'accord

- Temps de calcul pour les quatre algorithmes :

	n	40	2^{25}	2^{28}	2^{31}
`fibo1`	$\Theta(\varphi^n)$	31s	calcul irréalisable		
`fibo2`	$\Theta(n)$	0s	18s	`Segmentation fault`	
`fibo3`	$\Theta(n)$	0s	4s	25s	195s
`fibo4`	$\Theta(\log(n))$	0s	0s	0s	0s

Méthodes de programmation

Algorithme glouton

- Un algorithme glouton est un algorithme itératif qui à chaque étape essaye de s'approcher au maximum de la solution.
- Par exemple :
  - ▷ le rendu de monnaie : quelle somme de billets/pièces permet d'atteindre une valeur donnée ?
    Pour rendre 37 euros, on calcule $37 = 20 + 10 + 5 + 2$.
    ⇾ les valeurs des billets/pièces font que la solution est optimale.
  - ▷ le problème du voyageur de commerce : quel est le plus court chemin passant par n villes données ?
    On va toujours à la ville la plus proche non encore visitée.
    ⇾ donne une bonne solution, mais rarement optimale.

Programmation dynamique

- La programmation dynamique consiste à stocker les résultats des instances d'un problème que l'on résoud pour ne jamais avoir à résoudre deux fois une même instance.
  - ▷ permet parfois de rendre efficace un algorithme récursif simple.

- Par exemple :
  - ▷ suite doublement récursive : en stockant les valeurs intermédiaires de `fibo1` on obtient `fibo2`.
    - → passe de $\Theta(2^n)$ à $\Theta(n)$ en utilisant un peu de mémoire.
  - ▷ coefficients binomiaux avec le triangle de Pascal
    - → voir TD 04.

Quelques autres méthodes

- Force brute : nom donné à un algorithme qui trouve la solution en essayant toutes les solutions possibles.
  - ▷ recherche de clef en cryptographie.

- Diviser pour régner : méthode récursive qui consiste à diviser un problème en sous-problèmes, résoudre les sous-problèmes et recombiner les résultats.
  - ▷ algorithmes de tri (voir cours 05).

- Algorithmes génétiques : algorithme pour trouver une solution (quasi-)optimale d'un problème d'optimisation en partant de solutions approchées, en les faisant évoluer et en les sélectionnant.
  - ▷ un des projets d'IN104.

Théorie de la complexité

Qu'est-ce que la théorie de la complexité ?

- A pour but de classifier des problèmes en fonction de la complexité du meilleur algorithme pour les résoudre.
  - ⚠ il s'agit du meilleur algorithme, pas du meilleur connu...

- Pour Fibonnaci, nous avons un algorithme en temps $\Theta(\log n)$. Peut-on faire mieux ?
  - ▷ difficile à savoir, mais on a déjà une solution polynomiale :
    - → c'est un problème facile.
  - ▷ pour certains problèmes, on peut prouver des choses :
    - → un tri par comparaison coûte au moins $\Theta(n \log n)$.

Quelques définitions

Problème : question comportant un ou plusieurs paramètres.
- ▷ Quel est le plus court chemin entre deux sommets donnés d'un graphe ?

Instance : donnée du problème sur une valeur de ses paramètres.
- ▷ $G = (S, A)$, $x, y \in G$; quel est le plus court chemin entre x et y dans G ?

Problème de décision : la solution du problème $\in \{\text{oui, non}\}$.
- ▷ Existe-t-il un chemin de longueur $\leq k$ donnée entre deux sommets d'un graphe ?

Problème de calcul : calculer la solution d'un problème.
- ▷ Calculer le plus court chemin entre deux sommets donnés d'un graphe.

Classes de complexité

- Les deux classes de problèmes dont on parle le plus sont :

Classe P : ensemble des problèmes de décision résolubles en temps polynomial

Classe NP : ensemble des problèmes de décision dont la solution peut être vérifiée en temps polynomiale quand la réponse est oui.

- Clairement, $P \subseteq NP$, mais :

$$P \stackrel{?}{=} NP$$

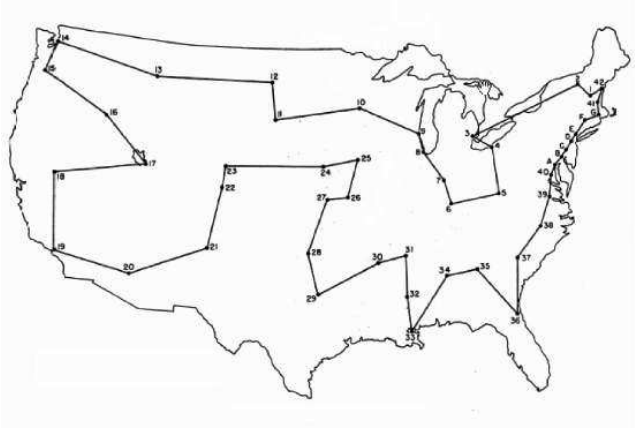
Note : beaucoup d'autres classes existent ($PSPACE$, $EXPTIME$, co-NP...)

Classes de complexité

- Un problème NP-complet est un problème dans NP au moins aussi difficile que tout autre problème de NP.
  - ▷ il existe une réduction polynomiale qui permet de récrire n'importe quelle instance de n'importe quelle problème de NP comme une instance de ce problème,
  - ▷ si on sait résoudre toutes les instances de ce problème on sait résoudre toutes les instances de tous les problèmes de NP.
- Si un problème NP-complet est dans P, alors $P = NP$.

- Exemples de problèmes NP-complets :
  - satisfiabilité d'une formule logique (SAT)
  - sac à dos
  - voyageur de commerce
  - 3-coloriage...

Un problème NP-complet
Le problème du voyageur de commerce



- Existe-t-il un parcours de n villes données de distance totale $\leq k$?
  - ▷ $n!$ parcours possibles,
  - ▷ vérifiable en temps polynomial, si on donne le parcours.

Problèmes dans P

- Tout ce que nous avons vu jusqu'à présent (ou presque) !
  - pgcd
  - algèbre linéaire
  - tri...

Test de primalité : étant donné un entier n, est-il premier ?

- Premier test polynomial trouvé en 2002 [Agrawal, Kayal, Saxena]
- Algorithme AKS, pas utile en pratique :
  - ▷ complexité en $O\left(\log(n)^{12}\right)$,
  - ▷ algorithmes probabilistes beaucoup plus efficaces.

Un problème dans NP

La factorisation

Factorisation : étant donné un entier n, trouver un diviseur de n différent de 1 et n ?

- On peut vérifier la solution en temps polynomial $\rightarrow\triangleright$ dans NP.
- Pas d'algorithme de factorisation polynomial connu. Au mieux :
  - ▷ crible quadratique : $O\left(2^{(\log n)^{\frac{1}{2}}(\log\log n)^{\frac{1}{2}}}\right)$,
  - ▷ crible algébrique : $O\left(2^{1.923\times(\log n)^{\frac{1}{3}}(\log\log n)^{\frac{2}{3}}}\right)$,
- Problème important en cryptographie :
  - ▷ une solution polynomiale serait embêtante...
  - ▷ prouver que c'est NP-complet serait très intéressant.

Challenges de factorisation

```
RSA-640

Prize: $20,000

Status: Factored (Nov. 2, 2005)

Decimal Digits: 193

310741824049004372135075003588856793003734602284272754572016194882320644051808150455634682967172328678243791627283803341547107310850191954852900733772482278352574238645401469173660247765234660 9
```

- Résolu en l'équivalent de 30 ans de calcul sur un Opteron 2.2GHz.

Challenges de factorisation

```
RSA-704

Prize: $30,000

Status: Not Factored

Decimal Digits: 212

740375634795617128280467960974295731425931888892312890849362326389727650340282662768919964196251178439958943305021275853701189680982867331732731089309005525051168770632990723963807867100860969625379346505637963596359
```