Introduction à l'algorithmique et la complexité (et un peu de CAML)

Complexité temporelle des algorithmes

Nicolas Nisse

Université Côte d'Azur, Inria, CNRS, I3S, France

Cours dispensés en MPSI (option Info) au CIV, depuis 2011-

http://www-sop.inria.fr/members/Nicolas.Nisse/lectures/prepa/

Outline

Comment mesurer l'efficacité d'un algorithme ?

Ce cours veut (espère) vous apprendre à évaluer l'efficacité des algorithmes et à concevoir des algorithmes efficaces !!
Lorsqu'on demande : "Mais c'est quoi un algorithme efficace ???", les réponses sont généralement:

- un algorithme qui "prend" le **moins de temps** ;
- un algorithme qui réalise le **moins d'opérations** ;
- un algorithme qui utilise le **moins de mémoire**.

Discutons de ces réponses :

- le "temps" (en secondes/minutes/heures...) d'exécution d'un algorithme *dépend de la machine sur lequel il est exécuté* (sur votre portable, ça ira sûrement moins vite que sur des "super-ordinateurs" avec plein de RAM et de multi-processeurs...). Cela *dépend aussi du langage de programmation* : un algorithme traduit en C++ ira probablement plus vite qu'un algorithme codé en Python... Bref, le "temps" n'est pas une mesure objective pour représenter la qualité d'un algorithme.
- **"le nombre d'opérations"** : c'est une très bonne idée, mais c'est quoi une opération ??? Par ailleurs, les opérations réalisées par un algorithme dépendent de ce à quoi l'algorithme est appliqué ??? Nous allons préciser tout ça dans la suite du cours.
- "le moins de mémoire" : là encore, c'est une bonne idée. On appelle ça la complexité spatiale. Cependant, nous ne parlerons pas de cet aspect dans ce cours (le temps manque, on peut pas parler de tout :( ).

Exemple du dictionnaire “français-espagnol" (1/2)

Considérons le problème qui, étant donné un dictionnaire $\mathscr{D}$ (disons “français-espagnol"), consiste à traduire un mot M (du français à l'espagnol). *entrées du pbm. : $\mathscr{D}$ et M.*

1^er essai de définition de complexité : Comme **mesure “d'efficacité"** des algorithmes pour résoudre ce problème, nous allons compter le nombre de mots du dictionnaire qu'il faut lire avant de finalement trouver la traduction de M. La complexité est le nombre d'opérations : ici, le nombre de mots lus avant traduction.

Étudions la complexité de l'algorithme suivant:

Algo 1 : Commencer par le premier mot de $\mathscr{D}$; tant qu'on ne rencontre pas le mot M, on passe au mot suivant ; lorsqu'(enfin) on trouve M, on traduit.

Quelle est la complexité (nombre de mots lus) de cet algorithme ?
Si on veut traduire $M = abaisser$, il faudra sûrement lire une dizaine de mots pour le traduire en “bajar". Pour traduire $M = zigzaguer$, il faudra lire presque tous les mots de $\mathscr{D}$ pour le traduire en “zigzaguear". Alors, quelle est la réponse ??? On considère le PIRE CAS : la complexité la plus grande parmi toutes les entrées possibles !

2^me essai de définition de complexité : La complexité est le nombre d'opérations dans le pire cas (pour la pire entrée).

Mais alors, quelle est la complexité d'Algo 1 ? Le “pire cas" est donc de traduire le dernier mot de $\mathscr{D}$... mais donc, ça dépend du nombre de mots dans $\mathscr{D}$ (de la “taille" de $\mathscr{D}$)??

Exemple du dictionnaire “français-espagnol" (2/2)

Définition de complexité : La complexité est le **nombre d'opérations** dans le **pire cas** (pour la pire entrée) **en fonction de la taille de l'entrée**.

Dans le cas de l'Algo 1, c'est donc la “taille" de $\mathscr{D}$ (nombre de mots dans $\mathscr{D}$).

Essayons un autre algorithme (celui que vous utilisez naturellement), **comparons le à Algo 1.**

Algo 2 (dichotomie) : Prenons le mot M' du mileu de $\mathscr{D}$. Si $M' = M$, on traduit le mot. Sinon, si $M < M'$ (dans l'ordre alphabétique/lexicographique), on recommence avec la première moitié de $\mathscr{D}$. Sinon ($M > M'$), on recommence avec la seconde moitié de $\mathscr{D}$.

Qui de Algo 1 ou Algo 2 est le plus efficace ?
Si on veut traduire (encore) $M = abaisser$, Algo 1 est plus rapide que Algo 2... et pourtant... quelle est la complexité de Algo 2 ?

Prenons $\mathscr{D}$ avec n mots. Dans le pire cas, on a vu que Algo 1 doit lire n mots. Pour Algo 2, à chaque mot lu, on “élimine" la moitié du dictionnaire. Quel que soit le mot M recherché, on ne lira qu'au plus $\lceil \log n \rceil$ mots (si vous ne voyez pas pourquoi... on y reviendra dans la suite du cours, cf. Slide 13).

Un algorithme A sera plus efficace (en pire cas) qu'un algorithme A' si la complexité (pire cas) de A est moindre que la complexité (pire cas) de A'.
Dans notre exemple, Algo 2 est donc plus efficace que Algo 1 (même si certaines entrées (ex: “abaisser") sont résolues plus efficacement par Algo 1 que par Algo 2).

Complexité temporelle

On mesure la complexité temporelle d'un algorithme en comptant le nombre d'opérations élémentaires réalisées par cet algorithme.

Opérations élémentaires : Par défaut, il s'agit de chaque opération arithmétique $(+,-,/,*)$, affectation ou comparaison. En général, le **contexte** précisera quelles opérations sont à considérer (par ex., compter seulement le nombre de comparaisons).

Étant donnés un problème $\mathscr{P}$, une instance (entrée) $\mathscr{I}$ du problème et un algorithme $\mathscr{A}$, soit $c(\mathscr{A},\mathscr{I})$ le nombre d'opérations élémentaires effectuées par $\mathscr{A}$ pour résoudre $\mathscr{P}$ sur l'instance $\mathscr{I}$. Notons que $c(\mathscr{A},\mathscr{I})$ est généralement une fonction de la taille $|\mathscr{I}|$ de l'instance $\mathscr{I}$.

Complexité en pire cas d'un algorithme $\mathscr{A}$: $c(\mathscr{A}) = \max\{c(\mathscr{A},\mathscr{I}) \mid \mathscr{I} \textit{ instance de } \mathscr{P}\}$.

Complexité du problème $\mathscr{P}$: $c(\mathscr{P}) = \min\{c(\mathscr{A}) \mid \mathscr{A} \textit{ algorithme pour } \mathscr{P}\}$.

La complexité en pire cas est “un peu pessimiste" (le “pire cas" est peut-être pathologiquement compliqué alors que la plupart des cas peuvent être résolus efficacement), il peut être plus intéressant (mais souvent plus compliqué) de considérer la complexité en moyenne.

(il ne sera que très peu question de cela dans ce cours)

Soit I l'ensemble de toutes les instances (entrées) possibles pour le problème $\mathscr{P}$.

Complexité en moyenne d'un algorithme $\mathscr{A}$: $c_{moy}(\mathscr{A}) = \sum_{\mathscr{I}\in I} c(\mathscr{A},\mathscr{I})/|I|$.

Notation "O"

Considérons un algorithme $\mathscr{A}$ pour résoudre un problème $\mathscr{P}$.
Comme nous l'avons dit, étudier la complexité de $\mathscr{A}$ revient à déterminer, pour chaque instance (entrée) $\mathscr{I}$ du problème, le nombre (**en fonction de la taille n de $\mathscr{I}$**) "d'opérations élémentaires" effectuées par $\mathscr{A}$ pour résoudre $\mathscr{P}$ pour l'instance $\mathscr{I}$ (et déterminer le pire cas : une instance qui requière le plus grand nombre d'opérations).

En fait, ce qui nous intéresse, c'est :

"l'ordre de grandeur", en fonction de n, du nombre "d'opérations élémentaires".

La complexité d'un algorithme sera (sauf mention contraire) exprimée sous la forme : $O(f(n))$.

Rappel rapide : Notation "O"

Soient $f = (f_n)_{n\in\mathbb{N}}$ et $g = (g_n)_{n\in\mathbb{N}}$, deux fonctions $\mathbb{N} \to \mathbb{R}$. On note $f = O(g)$ ssi $\exists c \in \mathbb{R}^+$, $\exists n_0 \in \mathbb{N}$, tels que $\forall n \geq n_0$, $|f(n)| \leq c \cdot |g(n)|$.

On note $f = \Omega(g)$ ssi $g = O(f)$.

Exemples (à vous de les prouver)

$2n^4 + 10^{10}n^2 + n^{4/5} + 47 = O(n^4)$; $4124 + 0.000001 \log n = O(\log n)$;
$15\sqrt{n} + \log^{12} n = O(\sqrt{n})$; $134n^{367} = O(0.00001^n)$.

On note $f = O(1)$ si $f = O(g)$ avec $g(n) = 1$ pour tout $n \in \mathbb{N}$. C'est-à-dire, $f = O(1)$ si $\exists c \in \mathbb{R}^+$ tel que $|f(n)| \leq c$ pour tout $n \in \mathbb{N}$ (si f est "bornée" par une constante).

Outline

Somme des *n* premiers entiers

Nous avons déjà vu (et prouvé) ces algorithmes. Étudions leur complexité **c(algo)**.

```
>       Caml Light version 0.80

#let sommeIteratif n =
        let somme_courante = ref 0 in
        for i = 1 to n do
                somme_courante := !somme_courante + i
        done;
        !somme_courante ;;
sommeIteratif : int -> int = <fun>
#
```

Comptons : d'abord, une affectation, puis pour chaque itération de la boucle : une addition et une affectation. C-à-d.

$c(\mathit{sommeIteratif}(n)) = 1 + \sum_{i=1}^{n}(1+1) = 1 + 2n = O(n)$.

```
>       Caml Light version 0.80

#let rec sommeRecursif n =
        match n with
        | 0 -> 0;
        | n -> n + sommeRecursif (n-1);;
sommeRecursif : int -> int = <fun>
#sommeRecursif 18;;
- : int = 171
```

Par récurrence : $c(\mathit{sommeRecursif}(0)) = O(1)$;
Pour $n > 0$, une comparaison, puis une addition et l'exécution de $\mathit{sommeRecursif}(n-1)$.

$c(\mathit{sommeRecursif}(n)) = 1 + 1 + c(\mathit{sommeRecursif}(n-1))$

Donc : $c(\mathit{sommeRecursif}(n)) = 1 + 2n = O(n)$

On ne s'intéresse généralement qu'à l'ordre de grandeur (grand "O") de la complexité !!

Au passage, on voit qu'écrire un même algorithme de manière itérative ou récursive ne change *a priori* pas la complexité.

Est-ce que vous pouvez faire mieux pour calculer la somme des *n* premiers entiers ?

Somme des n premiers entiers

Nous avons déjà vu (et prouvé) ces algorithmes. Étudions leur complexité **c(algo)**.

```
>       Caml Light version 0.80

#let sommeIteratif n =
        let somme_courante = ref 0 in
        for i = 1 to n do
                somme_courante := !somme_courante + i
        done;
        !somme_courante ;;
sommeIteratif : int -> int = <fun>
#
```

Comptons : d'abord, une affectation, puis pour chaque itération de la boucle : une addition et une affectation. C-à-d.

$c(\textit{sommeIteratif}(n)) = 1 + \sum_{i=1}^{n}(1+1) = 1+2n = O(n).$

```
>       Caml Light version 0.80

#let rec sommeRecursif n =
        match n with
        | 0 -> 0;
        | n -> n + sommeRecursif (n-1);;
sommeRecursif : int -> int = <fun>
#sommeRecursif 18;;
- : int = 171
```

Par récurrence : $c(\textit{sommeRecursif}(0)) = O(1)$;
Pour $n > 0$, une comparaison, puis une addition et l'exécution de $\textit{sommeRecursif}(n-1)$.

$c(\textit{sommeRecursif}(n)) = 1+1+c(\textit{sommeRecursif}(n-1))$

Donc : $c(\textit{sommeRecursif}(n)) = 1+2n = O(n)$

On ne s'intéresse généralement qu'à l'ordre de grandeur (grand "O") de la complexité !!
Au passage, on voit qu'écrire un même algorithme de manière itérative ou récursive ne change *a priori* pas la complexité.
Est-ce que vous pouvez faire mieux pour calculer la somme des n premiers entiers ?

Let somme $n = n*(n+1)/2$;; complexité : $O(1)$!!!

Outline

Évaluation de Polynômes, Méthode de Horner

Rappel : le degré du polynôme est $O(n)$. (précisément, ci-dessous, $n-1$)

```
>       Caml Light version 0.80

#let evalPolynom tab y =
        let n = vect_length tab in
        let resultat_courant = ref 0 in
        for i = 0 to n-1 do
                let puiss = ref 1 in
                for j = 0 to i do
                        puiss := !puiss * y
                done ;
                resultat_courant := !resultat_courant + (tab.(i) * !puiss)
        done;
        !resultat_courant ;;
evalPolynom : int vect -> int -> int = <fun>
#
```

calcul de "y^i"

ajout de "a_i * y^i"

Algorithme "naïf" (slide 11, Chap. 2.)

Comptons : d'abord, 2 affectations, puis pour chaque itération de la boucle "i" : une affectation puis une boucle "j" de $(i+1)$ itérations avec : une affectation et une multiplication à chaque itération, puis une affectation, une addition et une multiplication. C-à-d.

$c(evalPolynom(n)) = 2 + \sum_{i=0}^{n-1} (1 + (\sum_{j=1}^{i} 2) + 3) = O(n^2)$.

L'objectif de ce cours est d'étudier (et apprendre à concevoir) des algorithmes "efficaces" (avec la complexité la plus petite). **L'algorithme de Horner** (ci-dessous) est "meilleur" que l'algorithme "naïf" précédent.

Algorithme de Horner :

```
>       Caml Light version 0.80

#let Horner tab y =
        let n = vect_length tab in
        let resultat_courant = ref tab.(n-1) in
        for i=2 to n do
                resultat_courant := y*(!resultat_courant) + tab.(n-i);
        done;
        !resultat_courant;;
Horner : int vect -> int -> int = <fun>
```

Terminaison : j'espère qu'elle est évidente pour vous...

Correction : invariant de boucle : après l'itération i,

$resultat\_courant = \sum_{j=n-i}^{n-1} tab.(j) y^{n-i+j}$

Complexité : $c(Horner(n)) = 2 + \sum_{i=2}^{n} 3 = O(n)$.

Pour évaluer un polynôme de degré n, il faut "au moins" lire ses $n+1$ coefficients. Donc l'algorithme de Horner, de complexité $O(n)$, est optimal.

Outline

Exponentiation : calcul de x^c

Considérons le problème qui prend $x \in \mathbb{R}$ et $c \in \mathbb{N}$ en entrées, et veut calculer x^c.

```
>       Caml Light version 0.80

#let exponentiation n c =
        let produit_courant = ref 1 in
        for i = 1 to c do
                produit_courant := !produit_courant * n
        done;
        !produit_courant ;;
exponentiation : int -> int -> int = <fun>
#
```

Algorithme itératif "naïf" (slide 8, Chap. 2.)

$$c(exponentiation(n,c)) = 1 + \sum_{i=1}^{c} 2 = O(c).$$

```
>       Caml Light version 0.80

#let rec exponentiationRec n c =
        match c with
        | 0 -> 1
        | c -> n * (exponentiationRec n (c-1));;
exponentiationRec : int -> int -> int = <fun>
#exponentiationRec 10 5;;
- : int = 100000
```

Algorithme récursif "naïf" (slide 16, Chap. 2.)

$c(exponentiationRec(n,0)) = O(1)$.
$c(exponentiationRec(n,c))$

$$= 2 + c(exponentiationRec(n,c-1)) = O(c).$$

Pour faire mieux (plus rapide), nous proposons une illustration de la méthode dite "diviser pour régner" (voir le chapitre suivant pour plus de précisions).

Calcul de n^c à partir du calcul de $n^{\lfloor c/2 \rfloor} = n^k$.

- si c est pair ($c = 2k$), alors $n^c = (n^k)^2$.
- si c est impair ($c = 2k+1$), alors $n^c = n * (n^k)^2$.

Exponentiation rapide : calcul de x^c

```
>       Caml Light version 0.80

#let rec expRapide x c =
        match c with
        | 0 -> 1
        | c -> let y = expRapide x (c/2) in
                let z = y*y in
                if ((c mod 2) = 0) then z
                else x*z;;
expRapide : int -> int -> int = <fun>
```

Terminaison/Correction : laissées au lecteur (récurrence sur c + argument du slide précédent).

Complexité :

- Supposons d'abord que $c = 2^k$. Posons c_k le nombre d'opérations élémentaires pour calculer *expRapide x* 2^k.
  Alors $c_k \leq c_{k-1} + 3$. (application de *expRapide x* 2^{k-1}, une comparaison, et au plus 2 multiplications). Donc $c_k = O(k)$.
  Donc $complexity(expRapide\ x\ 2^k) = O(k) = \log_2(2^k)$.
- Déterminons $complexity(expRapide\ x\ c)$ pour c quelconque.
  Soit k tel que $2^k \leq c \leq 2^{k+1}$. Prouvons que
  $complexity(expRapide\ x\ 2^k) \leq complexity(expRapide\ x\ c) \leq complexity(expRapide\ x\ 2^{k+1})$
  Cette étape est généralement évidente (là, c'est le cas)
- Donc $O(k) \leq complexity(expRapide\ x\ c) \leq O(k+1)$.
  D'où $complexity(expRapide\ x\ c) = O(k) = O(\log c)$

> L'algo. d'exponentiation rapide calcule x^c en temps $O(\log c)$.

(les mêmes arguments permettent de prouver que l'algorithme par dichotomie trouve un mot en $O(\log n)$ essais dans un dictionnaire avec n mots (cf. Slide 5).)

Outline

Recherche dans un tableau

Soit $k \in \mathbb{N}$. Soit $\mathcal{T}$ l'ensemble des tableaux avec n éléments (des entiers non nuls $\leq k$). Étant donnés $tab \in \mathcal{T}$ et un entier $x \leq k$. La question est de déterminer la première position de x dans le tableau tab (ou décider que x n'est pas dans le tableau).

```
#let recherche tab x =
        let n = vect_length tab in
        let res = ref (-1) in
        let i = ref 0 in
        while (!res<0) && (!i<n) do
                if (tab.(!i) == x) then res:= !i;
        done;
        !res;;
recherche : 'a vect -> 'a -> int = <fun>
```

Terminaison Cet algorithme ne termine pas !! (sauf si $tab.(0) = x$). En effet, j'ai oublié d'incrémenter i donc l'algo compare toujours x à $tab.(0)$. Pour corriger, il faut ajouter $i := !i + 1$ juste avant “done;".

Correction laissée au lecteur

Complexité Dans le pire cas, x n'est pas dans le tableau et il faut tester chaque élément pour le savoir : $O(n)$.

Une fois n'est pas coutume, étudions la complexité en moyenne $c_{moy} = \dfrac{\sum_{tab \in \mathcal{T}} c(recherche(tab,x))}{|\mathcal{T}|}$.

Notons que $|\mathcal{T}| = k^n$. Pour $1 \leq i \leq n$, soit $t_i = (k-1)^{i-1}k^{n-i}$ le nombre de tableaux dont la première occurrence de x est à la i^{me} position, et soit $t_{n+1} = (k-1)^n$ le nombre de tableaux ne contenant pas x. En réorganisant la somme, on obtient

$$c_{moy} = \frac{1}{k^n}((\sum_{i=1}^{n} i \cdot t_i) + n \cdot t_{n+1}) = \frac{1}{k^n}((\sum_{i=1}^{n} i(k-1)^{i-1}k^{n-i}) + n \cdot (k-1)^n) = \frac{1}{k}(\sum_{i=1}^{n} i(\frac{k-1}{k})^{i-1}) + n \cdot (\frac{k-1}{k})^n.$$

Posons $f(y) = \sum_{i=0}^{n} y^i = \frac{1-y^{n+1}}{1-y}$ (pour $y \neq 1$), on note que $\sum_{i=1}^{n} i(\frac{k-1}{k})^{i-1} = f'(\frac{k-1}{k}) \sim_{n \to \infty} k^2$ (en notant que $(\frac{k-1}{k})^n \to 0$).

On en déduit que $c_{moy} = O(k)$.

Recherche dans un tableau trié (e.g., dictionnaire)

Enfin, si l'on suppose que le tableau est TRIÉ, un algorithme de recherche dichotomique permet de trouver x avec une complexité $O(\log n)$ en pire cas. (Le Chapitre suivant est dédié au tri).

```
#let recherche2 tabTrie x =
        let rec rechercheAux tabTrie x debut fin =
                if (fin<debut) then (-1)
                else
                        begin
                        let current = debut + (fin-debut)/2 in
                        if (x == tabTrie.(current)) then current
                        else
                                begin
                                if (x < tabTrie.(current)) then rechercheAux tabTrie x debut (current-1)
                                else rechercheAux tabTrie x (current+1) fin;
                                end;
                        end;
        in
        let n = vect_length tabTrie in
        rechercheAux tabTrie x 0 (n-1);;
recherche2 : 'a vect -> 'a -> int = <fun>
```

Prouvez la terminaison et la correction de l'algorithme *recherche*2.
Prouvez que sa complexité dans le pire cas est $O(\log n)$.

Pour cela, on pose $c(n)$ la complexité (pire cas) de l'algorithme pour un tableau de taille n. On note ensuite $u_k = c(2^k)$, puis il faut prouver que $u_k = O(1) + u_{k-1}$ et $u_0 = O(1)$. On en déduit que $u_k = O(k)$ et on conclut comme dans le cas de l'exponentiation rapide (slide 14) : pour $2^k \leq n < 2^{k-1}$, $u_k \leq c(n) < u_{k+1}$ et donc $c(n) = O(k) = O(\log n)$.