Sur la Complexité d'une Fonction Récursive

jpr, L1 Info & Math (Oct 2014)

Calculer la **complexité** d'une fonction consiste à trouver combien *coûte* un appel à cette fonction, en choisissant au préalable une *unité de mesure* (nombre d'opérations, espace mémoire consommé, temps de calcul). Quelques exemples :

• La factorielle :

```
(define (fac n)                  ; calcule n! pour n ≥ 0
  (if (= n 0)
      1
      (* n (fac (- n 1)))))
```

Si l'on s'intéresse au nombre de multiplications, l'appel à `(fac n)` demande n-1 multiplications. Or, on s'intéresse à ce qui se passe lorsque n est grand, donc on assimile n-1 à n et on dit que l'on a une *complexité d'ordre n* (linéaire, du premier degré en n) en le nombre de multiplications.

*N.B. Notez que la **complexité en temps** est bien plus élevée [le calcul exact pour n=20000 demande beaucoup plus que deux fois le temps de calcul pour n=10000], comme quoi il est essentiel de préciser ce que l'on mesure. La phrase « la complexité est en O(n) » n'a en soi aucun sens. Voir l'exercice 6.8.12 du livre de cours PCPS...*

• L'appartenance à une liste L [cours 6] :

```
(define (member? x L)             ; x ∈ L ?
  (cond ((empty? L) #f)
        ((equal? x (first L)) #t)
        (else (member? x (rest L)))))
```

Il s'agit d'une *analyse* de liste, on s'intéresse souvent au **nombre d'appels à la primitive `rest`**, ici durant l'exécution de `(member? x L)` sur une liste L de n éléments. Il se peut que l'on trouve l'élément tout de suite en tête [le *meilleur cas*] ou que l'on soit obligé d'aller jusqu'au bout de la liste [le *pire des cas*, en n coups]. On donne souvent le résultat de **la complexité dans le pire des cas**. On notera O(n) pour exprimer : linéaire dans le pire des cas. Notez que si l'on fait 2n étapes de calcul, on dira encore O(n) : on omet les constantes multiplicatives, sauf bien entendu si l'on veut procéder à une analyse plus fine, par exemple pour comparer deux algorithmes !

RESUME : Lorsqu'on dit qu'un algorithme est en O(f(n)), cela signifie que *dans le pire des cas*, il aura un coût [nombre d'opérations, temps de calcul, etc] proportionnel à f(n). Vous verrez en Maths, ou plus tard en cours d'Algorithmique, des définitions plus précises[1].

[1] Si vous êtes impatient, tapez dans Google "notations de Landau" et lisez l'article de la Wikipédia.

• **Le tri par insertion**

```
(define (tri-ins L)       ; retourne une copie triée croissante de L
  (if (empty? L)
      L
      (insertion (first L) (tri-ins (rest L)))))

(define (insertion x LT)  ; hypothèse : LT est déjà triée
  (cond ((empty? LT) (list x))
        ((<= x (first LT)) (cons x LT))
        (else (cons (first LT) (insertion x (rest LT))))))
```

L'analyse est ici un peu plus fine, et dans ce cas, les maths – domaine ultime de la rigueur – vont nous sauver. Notons c_n la complexité du tri d'une liste de longueur n. Comme il s'agit d'un problème de *construction* de liste, on mesure **le nombre de fois que la primitive `cons` a été appelée durant l'exécution de l'algorithme**. La technique consiste à lire la définition de la fonction, et à la traduire en un système d'équations sur la suite (c_n).

- Le coût de l'insertion d'un élément à sa place dans une liste triée LT de longueur n est clairement en O(n) puisque dans le pire des cas, l'insertion se fera en queue de liste.
- La première ligne du corps de la fonction `tri-ins` exprime que le tri d'une liste vide L est égal à L, sans utiliser `cons`. Donc coût = 0 :

$$\boxed{c_0 = 0}$$

- Dans le cas général on doit faire la *somme* de plusieurs coûts :

```
(insertion (first L) (tri-ins (rest L)))
```

O(n) 1 c_{n-1} 1

$$\boxed{c_n = c_{n-1} + O(n)}$$

Reste à résoudre ces équations. Dans ce cas il est facile [cf cours 6 page 19] de trouver la valeur exacte du terme général $c_n = \frac{n(n+1)}{2}$. En ne conservant que le terme de plus haut degré et en oubliant les constantes multiplicatives, il vient une **complexité en $O(n^2)$** : l'algorithme est quadratique [du second degré]. Ce n'est pas un bon algorithme de tri puisque les meilleurs ont une complexité en O(n log n), juste au-dessus de la complexité linéaire.

En général, les équations à résoudre sont souvent plus difficiles et seule une bonne habileté mathématique permet de trouver un *équivalent à l'infini du terme général* de la suite. On peut aussi utiliser des logiciels de calcul formel comme *Mathematica* qui sont souvent capables de tels exploits…

N.B. Plus généralement, la solution de $c_n = c_{n-1} + O\left(n^d\right)$ *est* $c_n = O\left(n^{d+1}\right)$ *La démonstration figure dans le livre de cours PCPS page 92.*

Quatre remarques

1. Dans le cas général du tri par insertion, nous avions dit que le coût de `(first L)` était 1. En réalité, ce coût ne dépend pas de L, il est constant. Le **coût constant** se note O(1). Donc par exemple O(1) + O(n) = O(n), d'accord ? Ces notations sont un peu dangereuses mais les matheux font sans cesse des abus de langage et d'écriture sans lesquels tout texte mathématique deviendrait pédant et illisible… Il est clair qu'on ne peut faire mieux qu'un algorithme en O(1) ; hélas il y en a peu de ce type ☺. Les *meilleurs* sont plutôt les algorithmes *logarithmiques*, de coût O(log n). Vous en connaissez au moins un exemple : le nombre de multiplications dans le calcul de x^n par dichotomie. Mais la plupart des algorithmes courants ont un coût entre O(log n) et $O(n^2)$.

2. Dans le cas d'une *construction* de liste, il se peut qu'il soit aussi nécessaire de procéder à une *analyse* simultanée d'une autre liste. Tiens, le tri par exemple ! On parcourt une liste et on produit une autre liste : algorithme de tranformation. Dans ce cas, il peut être intéressant de calculer à la fois le nombre d'appels à `rest` (pour l'analyse) et le nombre d'appels à `cons` (pour la construction), puis de prendre le plus grand des deux. Imaginez la fonction qui analyse une liste L et qui construit une liste `(mini maxi)` formée du plus grand et du plus petit élément de L, quelle serait sa complexité ? O(n) et non 2…

3. Nous ne ferons pas de choses plus compliquées que cela en complexité, en 1ère année ! Le calcul el plus délicat est peut-être celui de la complexité du tri-fusion (Exo TP 7.1).

4. Si vous passez outre à la remarque 3, sachez qu'un très bon livre d'Algorithmique est ***Algorithmique*** (par Cormen, Leiserson & Rivest, chez Dunod), qui est à la B.U. Il contient aussi les analyses de complexité. Il est un peu difficile mais accompagnera la totalité de vos études. Les cours (pdf et vidéos) de ce livre sont disponibles au MIT dans la zone *Open Courseware*. Tapez dans Google : *OCW algorithms 5503*. Vous pouvez même les télécharger à partir de *iTunes* (gratuit) pour les consulter en podcast sur un baladeur numérique. *Of course, you understand English, don't you ? And you are very motivated by computer science ☺…*

   Un autre livre qui intéressera forcément le matheux est ***Concrete Mathematics*** de Knuth et Patashnik. Donald Knuth est un grand théoricien de l'algorithmique, mathématicien de formation, et auteur du système TEX de composition des documents scientifiques (le livre de cours PCPS est entièrement rédigé en TEX).