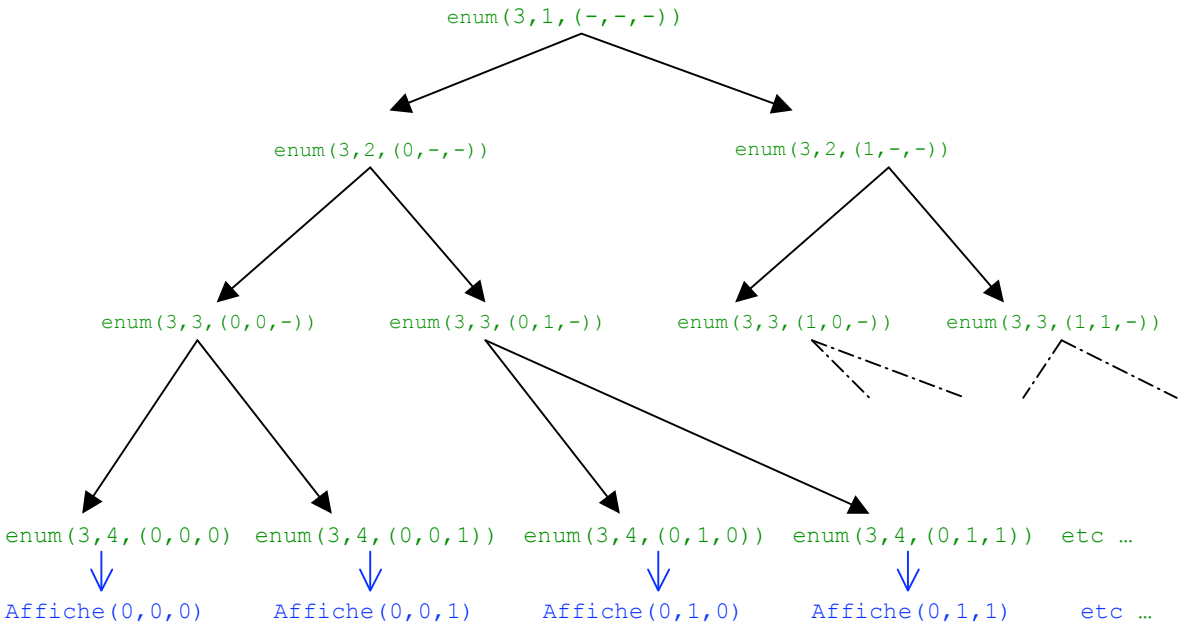
Corrigé E.D. Algorithmes et Structures de Données n° 1

Thème : Complexité des Algorithmes

Exercice I.1 De l'intérêt d'améliorer la taille des ordinateurs

Question 1

- Algo 1 affiche composantes du vecteur x. x ayant n composantes, la taille du problème est n. L'opération que l'on compte est `Afficher(`x_i`)` (c'est un choix arbitraire). Le nombre d'opérations est donc n.
- Algo 2 affiche la somme de toutes les paires de composantes du vecteur x. Ici aussi, la taille du problème est n. L'opération que l'on compte est `Afficher(`x_i+x_j`)`. Le nombre d'opérations est donc n^2.
- Algo 3 affiche la somme de tous les quintuplés de composantes du vecteur x. Ici encore, la taille du problème est n. L'opération que l'on compte est `Afficher(`$x_i+x_j+x_k+x_l+x_m$`)`. Le nombre d'opérations est donc n^5.
- Algo 4 affiche toutes les valeurs possibles des vecteurs de n composantes, toutes valant 0 ou 1. La taille du problème est n (car en entrée, on a un vecteur x de n composantes indéfinies). Ici, on considère arbitrairement que `Afficher(x)` est l'opération élémentaire qu'on compte. Cette opération est effectuée 2^n fois. On aurait pu considérer que l'opération à compter était l'affichage d'une composante de x. Dans ce cas `Afficher(x)` comporterait n opérations (cf. Algo 1) et Algo 4 compterait $n*2^n$ opérations. Voici l'arbre des appels générés par l'appel `enumeration(3,1,x)` :



Question 2 Compléter le tableau ci-dessous

Nombre d'opérations en fonction de la taille n des Données	Avec un ordinateur X		Avec un ordinateur Y 100 fois plus rapide que X	
	Taille maximale (n max) des problèmes traités en 1h	Nombre d'opérations Effectuées en 1h	Taille maximale (n max) des problèmes traités en 1h	Nombre d'opérations effectuées en 1h
N	N_1	N_1	$N'_1 = 100\ N$	N'_1
n^2	N_2	$(N_2)^2\ (=N_1)$	N'_2	$(N'_2)^2\ (=N'_1$
n^5	N_3	$(N_3)^5\ (=N_1)$	N'_3	$(N'_3)^5\ (=N'_1$
2^n	N_4	$2^{N4}\ (=N_1)$	N'_4	$2^{N'4}\ (=N'_1)$

Le but de cette question est d'exprimer N'_1 en fonction de N_1, N'_2 en fonction de N_2, N'_3, en fonction de N_3 et N'_4 en fonction de N_4.

- N'_1 en fonction de N_1

nbre d'opérations effectuées par Y en 1h = 100 x (nombre d'opérations effectuées par X en 1h) donc

$$\mathbf{N'_1 = 100\ N_1}$$

- N'_2 en fonction de N_2

En 1 heure, l'ordinateur Y effectue 100 fois plus d'opérations que l'ordinateur X. Or l'ordinateur X effectuait en 1 heure $(N_2)^2$ opérations (on avait $(N_2)^2 = N1$). Comme l'ordinateur Y effectue $(N'_2)^2$ opérations en 1h (ce qui implique $(N'_2)^2 = N'1$), on a:

Nb d'opérations effectuées par Y en 1h $= (N'_2)^2$
$= 100 \times$ nb d'opérations effectuées par X en 1h
$= 100 \times (N_2)^2$

Donc $(N'_2)^2 = 100 \times (N_2)^2$

On prend la racine carrée des deux membres (positifs) de l'égalité et on obtient :

$$N'2 = \mathbf{10}\ N2$$

- N'_3 en fonction de N_3

De la même façon :

Nb d'opérations effectuées par Y en 1h $= (N'_3)^5$
$= 100 \times$ nb d'opérations effectuées par X en 1h
$= 100 \times (N_3)^5$

Donc $(N'_3)^5 = 100 \times (N_3)^5$

On prend la racine cinquième des deux membres (positifs) de l'égalité et on obtient :

$$N'3 = \sqrt[5]{100}\ N3 = \mathbf{2{,}5}\ N3$$

- N'_4 en fonction de N_4

Rappel sur les logarithmes et les exponentielles :

$\log_2(x)$ est le logarithme à base 2 de x. $\log_2(x) = \frac{\ln(x)}{\ln(2)}$ où ln est le logarithme népérien.

On a : $\begin{vmatrix} 2^{\log_2(x)} = x \text{ (pour } x > 0) \quad \text{et } \log_2(2^x) = x \text{ (pour x réel)} \\ \log_2(xy) = \log_2(x) + \log_2(y) \end{vmatrix}$

Nb d'opérations effectuées par Y en 1h $= 2^{N'_4}$

$= 100 \times$ nb d'opérations effectuées par X en 1h

$= 100 \times 2^{N_4}$

Donc :

$2^{N'_4} = 100 \times 2^{N_4}$

Par passage au logarithme :

$\log_2(2^{N'_4}) = \log_2(100 \times 2^{N_4}) = \log_2(100) + \log_2(2^{N_4})$

Donc : $N'_4 = \log_2(100) + N_4$

$$N'_4 = 6 + N_4$$

Conclusion de l'exercice *: si on multiplie par 100 la rapidité de l'ordinateur, on ne multiplie pas par 100 la taille des problèmes que l'on peut traiter en un temps égal. D'où l'importance de trouver des algorithmes de faible complexité plutôt que de compter sur les progrès des processeurs pour certains types de problèmes !*

Exercice I.2

Dans les 3 algorithmes de l'exercice, on va compter le nombre d'opérations effectuées au cours de l'algorithme. Une "opération" peut être une affectation (p:=a_0), une addition (x+y), une multiplication (x*y) ou un test (si (a=2)). Ce choix est arbitraire : Souvent, on néglige les affectations lorsqu'on compte le nombre d'opérations. Il suffit de préciser au départ (comme ici, précisément) ce qu'on entend par "opération", pour savoir ce que l'on compte.

Algo 1)

Début

p:= a_0 — 1 affectation

pour i:=1 à n **faire**

calculer $p_i := x^i$

p:= p + a_i * p_i

fait

n fois { 1 affectation (i:= 1 à n) ; (i - 1) multiplications + 1 affectation ; + 1 add., 1 mult., 1 affectation }

fin

Nombre d'opérations f1(n) :

Nombre de multiplications pour calculer à chaque itération de la boucle pour x^i (c'est-à-dire x*x*...*x, i-1 fois) plus la multiplication $a_i * p_i$:

$$1+2+3+...+n = n*(n+1)/2$$

Autres opérations : 1 (l'affectation du départ) + 4n (les 3 affectations et l'addition dans la boucle).

Donc f1(n) = n*(n+1)/2 + 1 + 4n = (1/2) n^2 + (9/2) n + 1

L'algo 1 est donc en O(n^2).

Algo 2)

Début

p:= a_0; q:=1; — 2 affectations

pour i:=1 à n **faire**

q := q* x;

p:= p + a_i * q — n fois { 1 affectation ; 1 multiplication + 1 affectation ; + 1 add., 1 mult., 1 affectation }

fait

fin

Nombre d'opérations : f2(n) = 2 + n*6 = 6n +2

L'algo 2 est donc en O(n).

Algo 3)

Début

p:= a_n; — 1 affectation

pour i:=n à 1 **faire**

p:= p * x + a_{i-1} — n fois { 1 affectation + 1 multiplication ; + 1 affectation + 1 addition + 1 soustraction (i - 1 dans a_{i-1}) }

fait

fin

Nombre d'opérations : f3(n) = 1+ n*5 = 5n +1

L'algo 3 est donc en O(n).

Dans les calculs de complexité, en pratique, on fait le calcul plus grossièrement en négligeant les affectations et on étudie surtout ce qui se passe dans les boucles; les constantes (non fonction de n) sont souvent négligeables et il est peu important en général de savoir si c'est 12n ou 19n (sauf comparaison précise entre 2 algos comme les algos 2 et 3 ci-dessus).

Exercice I.3

Question 1 f(n) = 12n+7

Pour c, il faut une valeur supérieure à 12. Si on prend 13 (par exemple), on cherche alors n_0 tel que pour n supérieur à n_0, on ait 12n+7 ≤ 13n, soit n ≥ 7.

On pourrait aussi prendre n_0 = 1 et c = 19.

f(n) est en O(n).

Question 2 f(n) = $a_0 n^p + a_1 n^{p-1} + ... + a_{p-1} n + a_p$

On suppose a_0>0.

Si tous les coefficients sont positifs, on peut alors choisir :

$c = a_0 + a_1 + ... + a_p$, avec $g(n) = n^p$

On a bien $f(n) <= (a_0 + a_1 + ... + a_p)n^p$

Si des coefficients sont négatifs, il suffit pour c de faire la somme des coefficients positifs.

Donc f est en $O(n^P)$.

Exercice I.4

La complexité (le nombre d'opérations) dépend du nombre de fois où on passe dans la boucle tant que : à chaque passage on fait soit 6 ou 5 opérations (test, affectation, addition ou division), selon le résultat du test "L(milieu)<A", donc un nombre constant d'opérations (au plus 6, ça ne dépend pas de n, de "place" ni de f).

Soit I le nombre de passages dans la boucle. On va calculer I. A chaque fois on divise la taille de l'espace de recherche (f-place+1) en deux et on s'arrête quand il ne contient plus qu'un seul élément. Voici l'évolution du nombre d'éléments dans l'espace de recherche après chaque itération :

Itération	1	2	3	4	...	I
Taille de l'espace de recherche	$n/2$	$n/2^2$	$n/2^3$	$n/2^4$	...	$\mathbf{n/2^I = 1}$

"On a un seul élément à l'itération I" se traduit par $n/2^I = 1$.

Donc $n = 2^I$

$\log_2(n) = \log_2(2^I) = I$

$I = \log_2(n)$

L'algo 4 est donc en $O(\log_2(n))$.