Mesure des algorithmes : O-notation

Introduction

Efficacité = Temps + Espace

Soient 2 programmes différents pour calculer le terme U_n de la suite de fibonacci:

```
P1:   i:=1; j:=0;
      Pour k:=1,n
        j := i+j;
        i := j-i;
      Fp;
      // résultat dans j
```

```
P2 :  Fib(n:entier):entier
      Si n < 2
        Fib := 1;
      Sinon
        Fib := Fib(n-1)+Fib(n-2)
      Fsi
```

Le tableau suivant résume les temps d'exécution de P1 et P2 sur la même machine (Cyber835) avec différentes valeurs de n:

n	***10***	***20***	***30***	***50***	***100***
P1	1/6 ms	1/3 ms	1/2 ms	3/4 ms	1,5 ms
P2	8 ms	1 s	2 min	Estim > 21j	Estim $>10^9$ ans

Le temps d'exécution de P1 augmente linéairement avec n alors que celui de P2, il augmente de manière exponentiel. P2 a une complexité beaucoup plus forte que celle de P1.

Définition

- $O(f(n)) = \{ t:\mathbb{N}\rightarrow\mathbb{R}^* \ / \ \exists\ c \in \mathbb{R}^+, \ \exists\ n_0 \in \mathbb{N}, \ \forall\ n \geq n_0 : \ t(n) \leq c\ f(n) \}$

$\mathbb{N}$: l'ensemble des entiers naturels
$\mathbb{R}^*$: l'ensemble des réels positifs ou nuls
$\mathbb{R}^+$: l'ensemble des réels strictement positifs

O(f(n)) est l'ensemble de toutes les applications t inférieures à c*f(n)
f(n) est un majorant de l'ensemble.

Exemples :

- $n^2 \in O(n^2)$
- $n^2 \in O(n^3)$ car $n^2 <= c*n^3$ pour c=1 et n>=0
- $2^{n+1} \in O(2^n)$ car $2^{n+1} = 2*2^n <= c\ 2^n$ pour c>=2 et n >= 0
- $3^n \notin O(2^n)$ car $(3/2)^n$ est croissante donc il n'existe pas de constante >= $(3/2)^n$ $\forall n$
- $(n+1)! \notin O(n!)$ car $(n+1)!/n! = n+1$ qui est aussi croissante
- $\log_b n \in O(\log_a n)$ car $\log_b n=(1/\log_a b)*\log_a n$ et $(1/\log_a b)$ est une cste

- Comportement asymptotique : Si un algo A est de complexité O(f(n)), alors f(n) est le comportement asymptotique du temps d'exécution de A.
- Principe d'invariance : La complexité d'un algorithme est indépendante de son implémentation (choix de la machine, du langage, ...). En d'autres termes, deux implémentations différentes d'un même algorithme ne peuvent différer en efficacité que par une constante multiplicative près. c1*f(n) pour l'un et c2*f(n) pour l'autre.

Propriétés

a) $O(f(n)) = O(g(n))$ ssi $f(n) \in O(g(n))$ et $g(n) \in O(f(n))$
b) $O(f(n)) \subset O(g(n))$ ssi $f(n) \in O(g(n))$ et $g(n) \notin O(f(n))$
c) $O(f(n)+g(n)) = O(\max(f(n),g(n)))$
d) Si $\lim (f(n)/g(n)) = k$ $(k>0)$ Alors $O(f(n)) = O(g(n))$
e) Si $\lim (f(n)/g(n)) = 0$ Alors $O(f(n)) \subset O(g(n))$

Quelques complexités usuelles

O(log n)	-->	logarithmique	--> la plus faible
O(n)	-->	linéaire	
$O(n^2)$	-->	quadratique	
$O(n^k)$	-->	polynomiale (k une constante)	
$O(a^n)$	-->	exponentiel (a une constante)	--> la plus forte

Autre notation (la Oméga notation)
$\Omega(f(n))$: l'ensemble de toutes les applications t >= c*f(n)
f(n) est un minorant de l'ensemble

Opérations élémentaires

C'est une opération du programme dont le temps d'exécution est indépendant de la taille (n) de l'exemplaire du problème à résoudre.
Généralement toutes les opérations simples (affectation, E/S d'une unité de donnée, évaluation d'expression simples, ...) ainsi que les comparaison (<, >, ...) ont un temps d'exécution indépendant de la taille du problème à résoudre.

Règles de calcul de la complexité

- O(1) pour toute opération élémentaire
- La complexité d'une séquence de 2 modules M1 de complexité O(f(n)) et M2 de complexité O(g(n)) est égale à la plus grande des complexité des deux modules : O(max(f(n),g(n)))
- La complexité d'une conditionnelle (Si cond Alors M1 Sinon M2 Fsi) est le max entre les complexités de cond, M1 et M2 (cette simplification s'effectue dans une analyse en pire cas)
- La complexité d'une boucle est égale à la somme sur toutes les itérations de la complexité du corps de la boucle

Exemple : Donner la complexité du programme suivant:

```
1       Pour i:=1 , n-1
2               Pour j:=i+1 , n
3                       Si T[i] > T[j]
4                               tmp := T[i];
5                               T[i] := T[j];
6                               T[j] := tmp;
7                       Fsi
8               Fp
9       Fp
```

Chaque itération de la boucle interne (2-8) prend au maximum un temps T = a (une constante incluant le test de ligne 3, les trois affectations et les instructions de contrôle de la boucle:test incrémentation, ...)
La boucle interne fait n-i itérations, donc T = b + (n-i) a où b désigne le temps d'initialisation de la boucle.
Pour chaque itération de la boucle externe, le temps T = c + b + (n-i) a avec c une constante englobant le temps nécessaire aux instructions de contrôle de la boucle.
La boucle externe fait n-1 itération, donc son temps d'exécution T est (borné par) :

$= d + \text{Somme}_{\text{sur } i=1,n-1} [c + b + (n-i) a]$

$= d + (n-1) (c+b) + (n-1) n a / 2$

d étant une constante représentant le temps nécessaire à l'initialisation de la boucle externe.
Le temps d'exécution total est borné par $T = (a/2) n^2 + (c+b-a/2) n + (d-c-b)$
Donc le programme est en $O(n^2)$ car a/2 est positif.

Complexité des algo récursifs

→ Cela revient à résoudre une équation de récurrence
3 approches :

a) éliminer la récurrence par substitution de proche en proche
b) deviner une solution est la démontrer par récurrence
c) utiliser la solution de certaines équations connues

Application de a) « par substitution »

Soit la fonction récursive suivante :

```
Fact(n:entier) : entier
Si n <= 1
        Fact := 1
Sinon
        Fact := n * Fact(n-1)
Fsi
```

Posons T(n) le temps d'exécution nécessaire pour un appel à Fact(n)

T(n) est le max entre :

- la complexité du test n<=1 qui est en O(1)
- la complexité de : Fact := 1 qui est aussi en O(1)
- la complexité de : Fact := n*Fact(n-1) dont le temps d'exécution dépend de T(n-1) (pour le calcul de Fact(n-1))

On peux donc écrire que :
T(n) = a si n<=1 et,
T(n) = b+T(n-1) sinon (si n>1)

La constante 'a' englobe le temps du test (n<=1) et le temps de l'affectation (Fact := 1)
La constante 'b' englobe :

- le temps du test (n<=1)
- le temps de l'opération * entre n et le résultat de Fact(n-1)
- et le temps de l'affectation finale (Fact := ...)

T(n-1) (le temps nécessaire pour le calcul de Fac(n-1)) sera calculé (récursivement) avec la même décomposition.

Pour calculer la solution générale de cette équation, on peut procéder par substitution:

$T(n) = b + T(n-1) = b + [b+T(n-2)]$
$= 2b + T(n-2) = 2b + [b+T(n-3)]$
$= ...$
$= ib + T(n-i) = ib + [b+T(n-i+1)]$
$= ...$
$= (n-1)b + T(n-n+1) = nb - b + T(1) = nb - b + a$
$T(n) = nb - b + a$

Donc Fact est en O(n) car b est une constante positive.

Application de b) « par vérification »

Soit la procédure récursive du parcours inordre dans un arbre binaire :

```
Inordre( A:arbre )
Si A <> NIL
        Inordre( fg(A) );
        ecrire( Info(A) );
        Inordre( fd(A) );
Fsi
```

La taille du problème (n) est le nombre de noeud dans l'arbre de racine A.
Posons T(n) le temps nécessaire pour parcourir un arbre contenant n noeuds.
Supposons que le sous arbre gauche de A contienne n1 noeuds (donc le sous arbre droit doit renfermer n – n1 – 1 noeuds), nous pouvons alors écrire l'équation de récurrence suivante :

$T(n) = a$ si n=0 (le cas où A = NIL) et
$T(n) = b + T(n1) + T(n-n1-1)$ si n>0 (le cas où A <> NIL, avec n1 < n)
la constante 'a' représente le temps d'exécution du test (A<>NIL),
la constante 'b' représente le temps nécessaire pour test et l'exécution de la partie Sinon
T(n1) représente le temps d'exécution de Inordre(fg(A))
T(n – n1 - 1) représente le temps d'exécution de Inordre(fd(A))

Pour résoudre cette équation de récurrence, supposons que cette procédure soit en O(n)
c-a-d T(n) est de la forme : dn + c (avec d et c des constantes à déterminer)
vérifions cette hypothèse par récurrence :
T(0) = a d'après l'équation de récurrence et
T(0) = c d'après l'hypothèse de récurrence, donc il faut que c = a
$T(1) = b + T(0) + T(0) = b+2a$ d'après l'équation de récurrence (car pour n=1 les sous-arbres gauche et droit sont vides, c-a-d n1=0 et n-n1-1=0)
$T(1) = d + a$ d'après l'hypothèse de récurrence, donc d = b+a
l'hypothèse de récurrence est donc $T(n) = (b+a)n + a$
supposons cette hypothèse vraie jusqu'à m-1 et vérifions qu'elle reste vraie pour m:
$T(m) = b + T(m1) + T(m-m1-1)$ d'après l'équation de récurrence
comme m1 < m et (m-m1-1) < m on a alors :
$T(m1) = (b+a)m1 + a$ et $T(m-m1-1) = (b+a)(m-m1-1) + a$ d'après l'hypothèse de récurrence
donc $T(m) = b + [(b+a)m1 + a] + [(b+a)(m-m1-1) + a]$
$= b + (b+a)m1 + a + (b+a)m – (b+a)m1 - (b+a) + a$
$= (b+a)m + a$ donc l'hypothèse de récurrence est vraie quelque soit n

Application de c) « par identification à des équations connues »

1er cas : équations homogènes

Une équation de récurrence est dite homogène si elle a la forme suivante:

$$a_0t_n + a_1t_{n-1} + ... + a_kt_{n-k} = 0 \qquad (1)$$

pour trouver sa solution générale, on procède comme suit:
a) Etablir son équation caractéristique (un polynôme de degré k) :

$$a_0x^k + a_1x^{k-1} + ... a_k = 0$$

calculer ses racines: $r_1, r_2, ... r_k$

b) Si toutes les racine r_i sont distinctes, alors la solution générale de (1) est donnée par :

$$t_n = C_1r_1^n + C_2r_2^n + ... C_kr_k^n \qquad (2)$$

où les C_i sont des constantes que l'on peut déterminer avec les conditions initiales de l'équation de récurrence.

c) Si r_j est une solution multiple (de multiplicité m : apparaît m fois comme solution de l'équation caractéristique), alors la solution générale de (1) est donnée en remplaçant dans (2) le terme $C_j r_j^n$ par la somme : $C_{j1} r_j^n + C_{j2} n r_j^n + C_{j3} n^2 r_j^n \ldots C_{jm} n^{m-1} r_j^n$

Exemples:

1) Soit : $t_n - 3\, t_{n-1} - 4\, t_{n-2} = 0$ (forme générale)

$t_0 = 0$, $t_1 = 1$ (conditions initiales)

son équation caractéristique est : $x^2 - 3x - 4 = 0$ --> 2 racines distinctes : $r_1 = -1$ et $r_2 = 4$

donc la solution générale est : $t_n = C_1(-1)^n + C_2\, 4^n$ --> $O(4^n)$ car C_2 est positive

en appliquant les conditions initiales on trouve C_1=-1/5 et C_2=1/5

2) Soit : $t_n - 5\, t_{n-1} + 8\, t_{n-2} - 4\, t_{n-3} = 0$ (forme générale)

$t_0 = 0$, $t_1 = 1$, $t_2 = 2$ (conditions initiales)

son équation caractéristique est : $x^3 - 5x^2 + 8x - 4 = 0$ ou alors : $(x-1)(x-2)^2 = 0$

--> 3 racines : 1, 2 et 2 (2 est une solution multiple de multiplicité = 2)

donc la solution générale est : $t_n = C_1 1^n + C_2 2^n + C_3 n 2^n$

les conditions initiales donnent : $C_1 = -2$, $C_2 = 2$ et $C_3 = -1/2$

2e cas : équations non homogènes:

Une équation de récurrence est dite non homogène si elle a la forme suivante:

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = b_1^n P_1(n) + b_2^n P_2(n) + b_3^n P_3(n) + \ldots \qquad (3)$$

où les b_i sont des constantes distinctes et les $P_i(n)$ des polynômes en n de degré d_i

La résolution d'une telle équation suit le même schéma que celui des équations homogènes en partant de l'équation caractéristique suivante :

$$(a_0 x^k + a_1 x^{k-1} + \ldots a_k)\,(x-b_1)^{d1+1}\,(x-b_2)^{d2+1}\,(x-b_3)^{d3+1} \ldots = 0$$

Exemple:

Soit : $t_n - 2t_{n-1} = n + 2^n$ et $t_0 = 0$

c'est une équation non homogène avec :

b1 = 1 , P1(n) = n , d1 = 1

b2 = 2 , P2(n) = 1 , d2 = 0

son équation caractéristique est donc : $(x-2)(x-1)^2(x-2) = 0$ dont les racines sont : 2, 1, 1, 2

donc les racines 1 et 2 sont, chacune, de multiplicité 2

La solution générale est alors donnée par : $t_n = C_1 1^n + C_2 n 1^n + C_3 2^n + C_4 n 2^n$