COMPLEXITÉ ET ALGORITHMS

Comparaison de croissance

$\lim_{n\to\infty} \frac{(\log n)^\alpha}{n^\beta} = 0$, $\forall \alpha, \beta > 0$

Echelle

$1 << \log n << (\log n)^c << n^\epsilon << n << n^c << c^n << n! << n^n << c^{c^n}$

Complexité

On distingue plusieurs type d'analyse de complexite. On étudie ici l'analyse des pire des cas. Si $T(A)$ est le nombre d'instruction nécessaire pour que l'algo fasse les calculs sur l'entrée A, on s'interesse à la suite (t_n) définie par

$$t_n = maxT(A),\ |A| = n$$

ou $|A|$ est la taille de l'entrée A. On se contentera d'éstimer t_n avec une ordre de grandeur O.
Un resultat typique, la complexité de l'algorithme de t_n par insertion est $O(n^2)$.

Proposition

Soient $\alpha, \beta > 0$, On a

$$\sum_{i=1}^{n} i^\alpha (\log i)^\beta = O(n^{\alpha+1} \log(n)^\beta)$$

Algo recursif

On commence par transformer l'analyse de l'algo en formule. On dit que $T(n)$ est le temps d'éxecution dans le pire des cas pour une données de taille "n" et on essai d'éxprimer $T(n)$ en fonction de $T(i)$.

Algo (Fibonacci)

Fiboncci(n)
Si $n \leq 1$ alors :
retourner 1
Sinon:
retourner Fibonacci(n-1) + Fibonacci(n-2)
fin
fin

On obtient l'équation: $T(n) = T(n-1) + T(n-2) + O(1)$ avec certaines conditions initiales en $T(0)$ et $T(1)$.
Le terme $O(1)$ correspond au temps pour éffectuer les instructions qui menent à faire des 2 appels recurssifs et pour éffectuer la somme des resultats.

Algo (Diviser pour regner)

La méthode de diviser pour regner est une méthode qui permet parfois de trouver des solutions éfficace à des problèmes algorithmiques. L'idée est de décomposer le problème initial de taille n, en plusieurs sous problème de taille sensiblement inferieur, puis de recombiner les solutions partielles.
L'exemple typique est l'algo de tri-fusion : pour trier un tableau de taille "n", on le decoupe en 2 tableaux de taille $\frac{n}{2}$ et l'etape de fusion permet de recombiner les 2 solution en $n-1$ operation.

Algo (Tri-fusion)

TriFusion(T):
Si $n \leq 1$:
retourner 1
Sinon:
n=|T|
$T_1 = TriFusion(T[0, \frac{n}{2}])$
$T_2 = TriFusion(T[\frac{n}{2}+1, n])$
retourner Fusion(T_1, T_2)
fin
fin

On va estimer la complexité en comptant le nombre T(n) de comparaison éffectue par l'algo.
On a $\begin{cases} T(0) & = & 0 \\ T(1) & = & 0 \\ T(n) & = & 2T(\frac{n}{2}) + n - 1 \end{cases}$
D'une manière générale, on aura:

Diviser: on decoupe le problème en "a" sous-problème de taille $\frac{n}{b}$ qui sont de même nature, avec $a \geq 1, b > 1$.

Regner: les problèmes sont resolus recurssivements

Recombiner: on utilise les solutions des sous-problèmes pour reconstruire la solution au problème initial en temps $O(n^d)$ avec $d \geq 0$

L'équation qu'on aura à resoudre est donc

$$\begin{cases} T(1) & = & constante \\ T(n) & \simeq & aT(\frac{n}{b}) + O(n^d) \end{cases}$$

Théorème (Fondamental)

On considère l'équation $T(n) = aT(\frac{n}{b}) + O(n^\lambda)$

1. Si $\lambda > d$ alors $T(n) = O(n^\lambda)$
2. Si $\lambda = d$ alors $T(n) = O(n^d \log n)$
3. Si $\lambda < d$ alors $T(n) = O(n^d)$

Ainsi, pour le cas de tri-fusion, on a $a = 2, b = 2, \lambda = d = 1$ donc on a une complexité $O(n \log n)$

Dichotomie

Si T est un tableau triee de taille n, on s'interesse a l'algorithme qui recherche si $x \in T$ au moyen d'une dichotomie.
Pour l'algo recursif, on spécifie un indice de début "d" et de fin "f" et on recherche si x est dans T entre les positions d et f (Initialement d=0 et f=n-1)

Algo (Dichotomie)

Recherche(T, x, d, f):
Si f < d:
retourner Faux
Sinon:
$m = \frac{a+b}{2}$
Si $T[m] = x$, retourner Vrai
Sinon si $T[m] < x$ alors reourner Recherche(T, x, m+1, f)
Sinon retourner recherche(T, x, d, m-1)
fin
fin
fin
fin

Exponentiation rapide

Il s'agit de calculer x^n pour x et n données en calculant la complexité par rapport a n. La méthode naïve (multiplions n fois 1 par x) donne une complexité linéaire. On peut faire mieux en utilisant le fait que

$$\begin{cases} x^0 &= 1 \\ x^n &= (x^2)\frac{n}{2} \text{ si n est pair} \\ x^n &= x(x^2)^{\frac{n-1}{2}} \text{ si n est impair} \end{cases}$$

Exponentiation rapide

Puissance(x, n):
Si $n = 0$, retourner 1
Sinon
Si n est pair, retourner Puissance(x*x, $\frac{n}{2}$)
Sinon, retourner x*Puissance(x*x, $\frac{n-1}{2}$)
fin
fin
fin

Algo de Karatsuba

On rappelle qu'un polynôme P est de la forme $P(X) = a_0 + a_1X + \cdots + a_nX^n = \sum_{i=0}^{n} a_iX^i$
Si $P = \sum_{i=0}^{n} a_iX^i$ et $Q = \sum_{i=0}^{n} b_iX^i$
Calculer $R = P + Q$ est facile car l'addition des polynômes revient a l'addition 2 a 2 des coefficient de même rang. On a ainsi

$$P + Q = (a_0 + b_0) + (a_1 + b_1)X + \cdots + (a_n + b_n)X^n$$

On peut donc le calculer em temps $O(n)$ en faisant un simple boucle. La multiplication est plus compliqué si on dévellope les premiers termes. On a

$$PQ = a_0b_0 + (a_0b_1 + a_1b_0)X + \cdots + a_nb_nX^n$$

La formule générale pour la k-eme coefficient C_k de PQ est $C_k = \sum_{i,j=k} a_ib_j$. Si on implemente cette règle en algorithme, on obtient une complexité $O(n^2)$. L'objectif est d'obtenir un multiplication plus rapide. Pour cela, on commence à décomposer P et Q en 2 polynôme. On ecrit $P = RX^{\frac{n}{2}} + S$; $Q = TX^{\frac{n}{2}} + U$ ou R, S, T, et U sont des polynôme de taille $\frac{n}{2}$. On peut muliplier les 2 expression et on obtient

$$PQ = RTX^n + (RU + ST)X^{\frac{n}{2}} + SU.$$

On peut effectuer les 4 produits $RT, RU, ST,$ et SU recurssivement puis on recombine en temps linéaire. On est dans un cas typique de diviser pour regner avec les paramètres $a = 4, b = 2$ et $d = 1$. Si on applique le théorème fondamental, on obtient une complexité $O(n^2)$. Pour ameliorer la complexité Karatsuba a remarqué que

$$PQ = RTX^n + ((R+S)(T+U) - (RT+SU))X^{\frac{n}{2}} + SU$$

On remarque qu'on a plus que 3 produit plus petit à effectuer RT, SU et $(R+S)(T+U)$. Le reste se fait en temps linéaire, on a donc le paramètre $a = 3, b = 2,$ et $d = 1$. En appliquant le théorème fondamental on a une complexité $O(n^{log_2 3}) = O(n^{1,585})$. On a gagné significativement en efficacité: