Diviser pour résoudre

Principe général

Diviser le problème de taille n en sous-problèmes plus petits de manière à ce que la solution de chaque sous-problème facilite la construction du problème entier. C'est une méthode descendante.

Chaque sous-problème est résolu en appliquant le même principe de décomposition (récursivement). Ce même procédé se répète jusqu'à obtenir des sous-problèmes suffisamment simples pour être résolus de manière triviale.

Le schéma général des algorithmes de type « diviser pour résoudre » est le suivant :

```
DPR(x) :                    // x est le problème à résoudre
Si x est suffisamment petit ou simple :
    . Retourner  A(x)       // A(x) un algo connu (peut être trivial) pour résoudre x
Sinon
    . Décomposer x en sous exemplaires x1,x2...xk
    . Pour i=1, k :         //
        yi := DPR(xi)       // résoudre chaque sous-problème récursivement
      Fp                    //
    . Combiner les yi pour former une solution y à x
    . Retourner y
Fsi
```

Pour pouvoir appliquer cette méthode, on doit :

- trouver une décomposition du pb en sous-pb plus simples
- avoir un algo A(x) capable de résoudre des pb simples
- savoir comment combiner les solutions intermédiaires pour former la solution globale.

Applications

1. Tri par fusion

Une technique "diviser pour résoudre" peut être la suivante:
Pour trier une liste de n éléments, il faut trier deux sous listes de n/2 éléments

chacune (les 2 moitiés de L) puis de faire l'inter-classement (ou fusion) entre ces deux listes. De la même façon, pour trier une liste de n/2 éléments il faut trier deux listes de n/4 éléments chacune puis de faire leur inter-classement. Et ainsi de suite jusqu'à l'obtention de listes d'un seul élément (dont le tri est trivial).
Ce qui peut s'énoncer de manière récursive comme suit:

```
Tri(L, n)
Si n = 1 :
    Tri := L                // L est déjà triée
Sinon
    L1 := L [1..n/2]        // récupérer la 1ere moitié en o(n)
    L2 := L [n/2+1..n]      // récupérer la 2e moitié en o(n)
    R1 := Tri( L1, n/2 )        // résolution des
    R2 := Tri( L2, n/2 )        // 2 sous pb.
    Tri := Fusion( R1,R2 )      // Combiner les 2 solutions.
Fsi
```

La fusion de 2 listes de n éléments se fait avec 2n itérations (donc o(n)).

Equation de récurrence associée :

T(n) = a si n =1
= 2T(n/2) + b n sinon

La constante 'a' désigne le temps nécessaire pour le test (n=1) et celui pour l'affectation (Tri:=L). Le terme 'bn' inclut le temps nécessaire pour le test, la récupération des 2 moitiés (boucle de n itérations) et la fusion des 2 solutions (boucle de 2n itérations).

Solution : O(n Log n) ce qui est meilleur que beaucoup d'algorithmes de tri en $o(n^2)$.

Si nous voulons construire une solution non récursive, en remontant dans les appels récursifs, on arrive à l'algorithme de VON-NEUMANN suivant :
Cela revient donc à considérer au départ n sous listes de 1 élément, donc triées. Ensuite on fusionne deux à deux, puis quatre en quatre. etc...

2. Multiplication de grands entiers

Soient X et Y deux grand nombres ayant n chiffres. (n suffisamment grand).
Pour calculer le produit de X par Y on peut programmer l'algorithme que l'on a appris en primaire et qui consiste à multiplier chaque chiffre de Y par tous les chiffres de X en sommant les résultat intermédiaire :

Posons $x_n\ x_{n-1} \ldots x_3\ x_2\ x_1$ les chiffres formant X.
et de même $y_n\ y_{n-1} \ldots y_3\ y_2\ y_1$ les chiffres représentant Y
La méthode habituelle demande n^2 multiplications entre chiffres et n additions de résultats intermédiaires avec des décalages vers la droite à chaque itération :

				x_n	x_{n-1}	...	x_3	x_2	x_1
			X	y_n	y_{n-1}	...	y_3	y_2	y_1
				---	---	---	---	---	---
				y_1*x_n	y_1*x_{n-1}		y_1*x_3	y_1*x_2	y_1*x_1
		y_2*x_n		y_2*x_{n-1}	y_2*x_3		y_2*x_2	y_2*x_1	0
	y_3*x_n	y_3*x_{n-1}		y_3*x_3	y_3*x_2		y_3*x_1	0	0
+	....			....		....	0	0	0
	---	---	---	---	---	---	---	---	---
=	...		....		....		....		....

Les additions et décalages demandes un nombre d'itérations proportionnel à n, donc cette méthode (habituelle) a une complexité de $O(n^2)$.

Une solution " Diviser pour résoudre" de ce problème est de scinder X et Y en deux entiers de n/2 chiffres chacun :
Soient A la première moitié de X (les n/2 chiffres de poids fort) et B la 2e moitié de X
Soient C la première moitié de Y (les n/2 chiffres de poids fort) et D la 2e moitié de Y

$A = x_n\ x_{n-1} \ldots x_{n/2}$ $B = x_{n/2+1} \ldots x_2\ x_1$ $C = y_n\ y_{n-1} \ldots y_{n/2}$ $D = y_{n/2+1} \ldots y_2\ y_1$

on peut écrire alors :

$X = A\ 10^{n/2} + B$ et $Y = C\ 10^{n/2} + D$

donc

$$X\ Y = (A\ 10^{n/2} + B)\ (C\ 10^{n/2} + D) = AC\ 10^{n} + (AD + BC)10^{n/2} + BD$$

4 multiplications de n/2 chiffres (AC, AD, BC, BD)
3 additions en o(n)
2 décalages (multiplication par 10^n et $10^{n/2}$) en o(n)

L'équation de récurrence d'un tel algo serait alors :

$T(1) = a$ si n=1
$T(n) = 4T(n/2) + b\ n$ sinon

On peut trouver la solution par substitution ou bien en utilisant la technique des équations différentielles en posant par exemple $n = 2^k$, ce qui donne l'équation de récurrence : $t_k - 4t_{k-1} = b\ 2^k$ dont la solution est connue $o(4^k)$ donc $o(n^2)$).
Cette solution (diviser pour résoudre) n'est pas plus efficace que la méthode habituelle

(même complexité).

En diminuant le nombre de multiplications (donc le nombre d'appels récursifs), on peut diminuer la complexité de l'algo diviser pour résoudre :

Remarquons que $X\,Y = AC10^{n} + [\,(A-B)(D-C) + AC + BD]10^{n/2} + BD$

3 multiplications (AC, BD, (A-B)(D-C))
4 additions en o(n)
2 soustractions en o(n) A-B et D-C
2 décalages en o(n)

$T(n) = 3T(n/2) + c\,n$

Solution $O(n^{1,59})$, avec $1,59 = \log_2 3$, ce qui est meilleur que $o(n^{2})$

<u>Remarques :</u>

Méthode asymptotiquement plus efficace mais beaucoup moins limpide de point de vue pédagogique. Efficace pour les grands nombres(>500 chiffres)

```
Fonction Mult(X, Y, n)
Si n = 1    retourner X*Y
Sinon
    A := n/2 premiers chiffres de X
    B := n/2 derniers chiffres de X
    C := n/2 premiers chiffres de Y
    D := n/2 derniers chiffres de Y
    S1 := moins(A,B);     // calcule A-B
    S2 := moins(D,C)      // calcule D-C
    m1 := Mult(A,C, n/2)              // calcule AC
    m2 =  Mult(S1,S2, n/2)            // calcule (A-B)(D-C)
    m3 := Mult(B,D, n/2)              // calcule BD
    S3 := plus( m1,m2 ) // calcule [AC] + [ (A-B)(D-C) ]
    S4 := plus( S3,m3 )  // calcule [ AC+(A-B)(D-C) ] + [BD]
    P1 := Décalage(m1, n)      // calcule  AC 10^n
    P2 := Décalage( S4 )       // calcule [AC+(A-B)(D-C)+BD] 10^n/2
    S5 := plus( P1,P2 )   // calcule  [AC 10^n] + [ (AC+(A-B)(D-C)+BD) 10^n/2 ]
    S6 := plus( S5,m3 )   // calcule [AC 10^n+(AC+(A-B)(D-C)+BD) 10^n/2 ] + [BD]
    retourner S6
Fsi
```

3. Organisation d'un tournoi circulaire :

Organisation d'un tournoi comprenant $n=2^k$ joueurs (ou équipe). Chaque joueur doit affronter chaque autre joueur une seule fois à raison d'un match par jour. Donc la durée du tournoi est de (n-1) jours.

Pour élaborer une solution pour n joueurs, on essaie d'élaborer une solution pour n/2 joueurs. De la même façon pour élaborer une solution pour n/2 joueurs, on essaie d'élaborer une solution pour n/4. Et ainsi de suite, jusqu'à atteindre 2 joueurs où la solution est triviale.

n = 2

	1	
1!	2	les indices de lignes représentent les joueurs
2!	1	les indices de colonnes représentent les jours

n = 4 = 2^2

	1	2	3	
1!	2!	3	4	(k=2)
2!	1!	4	3	
	---	---	---	
3!	4!	1	2	
4!	3!	2	1	

n = 8 = 2^3

	1	2	3	4	5	6	7	
1!	2	3	4!	5	6	7	8	(k=3)
2!	1	4	3!	6	7	8	5	
3!	4	1	2!	7	8	5	6	
4!	3	2	1!	8	5	6	7	
	---	---	---	---	---	---	---	
5!	6	7	8!	1	4	3	2	
6!	5	8	7!	2	1	4	3	
7!	8	5	6!	3	2	1	4	
8!	7	6	5!	4	3	2	1	

Ect....

Technique :

Pour k = 1 (ou n=2), la construction de la table est triviale.

A l'étape k ($n=2^k$):
- Partie supérieure gauche : Table de l'étape k-1.
- Partie inférieure gauche : Ajouter 2^{k-1} à chaque élément de la partie supérieure gauche.
- Partie supérieure droite : Au jour d'indice 2^{k-1} mettre les indices $2^{k-1}+1$, $2^{k-1}+2$, ... avec permutation circulaire (vers le haut) pour les autres jours.
- Partie inférieure droite : Au jour 2^{k-1} mettre les indices 1, 2, ... avec permutation circulaire (vers le haut) pour les autres jours.

4. Tour de Hanoi

Pour certains problèmes, il est beaucoup plus facile de rechercher une solution récursive plutôt qu'une itérative. Le problème des tours de Hanoi constitue un très bon exemple comme nous allons le montrer ci après.

Enoncé du principe.

On dispose de 3 tours (ou piquets) A, B et C. Initialement, n disques de diamètres différents sont placés sur A. Un disque plus grand ne doit jamais se trouver sur un disque plus petit. Le problème est de transférer les n disques de A vers C, en utilisant B comme intermédiaire en respectant les deux règles suivantes :
- à un moment donné, seul le disque au sommet d'un piquet peut être transféré vers un autre.
- un disque plus grand ne doit pas se trouver sur un disque plus petit.

Il est très difficile de trouver une solution itérative (essayer...). Par contre, il existe une solution récursive simple et élégante.

Supposons que nous avons une solution pour n-1 disques. Nous pourrons alors définir une solution pour n disques au moyen de la solution de n-1 disques. C'est une décomposition du pb du déplacement de n disques en 2 sous-pb de taille n-1 chacun.

n=1 constitue le cas trivial : transférer le disque unique de A vers C.
Pour transférer n disques de A vers C, en utilisant B comme auxiliaire, on procède comme suit:

1- Transférer les n-1 (premiers) disques du sommet A vers B en utilisant C comme intermédiaire.
2- Transférer le disque restant de A vers C.
3- Transférer les n-1 disques de B vers C avec A comme auxiliaire.

Nous remarquons que c'est simple car nous venons de développer une solution pour le cas trivial (n=1) et une solution pour le cas n en terme de solution pour le cas plus simple n-1.

L'algorithme est alors le suivant:

```
Procedure TourdeHanoi( n : entier; de, vers, aux : caractères);
Si n = 1   { cas trivial }
      "transfert du disque 1 de",       de, "vers", vers)
Sinon
      { Transférer les n-1 disques  au sommet de A vers B avec C comme auxiliaire }
      TourdeHanoi(n-1, de,aux,vers)

      { transférer le disque restant   de A vers C }
      "transférer le disque",      n,"de",de,"vers", vers)

      { Transférer les n-1 disques  de B vers C, utilisant A comme auxiliaire }
      TourdeHanoi(n-1, aux, vers,    de)
Fsi
```

- Complexité de TourdeHanoi
Si T(n) est le temps d'exécution d'un appel pour déplacer n disque, alors on peut écrire:
T(n) = a si n=1
T(n) = 2 T(n-1) + b sinon
C'est une équation non homogène $t_n - 2t_{n-1} = b$ d'équation caractéristique (x-2)(x-1)=0
Donc la solution générale est $T(n) = C1\ 2^n + C2$ donc en $O(2^n)$

Remarque :
Si nous voulons construire un algorithme non récursif, en remontant dans les appels, on aboutit à l'algorithme suivant :
Imaginons que les piquets sont disposés en triangle. A tous les coups de rang impair, on déplace le plus petit disque sur le piquet suivant dans le sens des aiguilles d'une montre.
A tous les coups de rang pair, on effectue le seul déplacement possible permis

n'impliquant pas le plus petit disque.

Recommandations générales sur « Diviser pour Résoudre »

1. Equilibrer les sous-problèmes

La division d'un problème en sous problème de taille sensiblement égales contribue de manière cruciale à l'efficacité de l'algorithme.

Considérer le tri par insertion.
On suppose T[1..K] trié et on insère T[K+1] par décalage

```
Tri(L, n) :
Si n = 1 :  retourner(L)
Sinon       Retourner(Insère[Tri(n-1),T(n)])  Fsi
```

Division du problème en deux sous problèmes de taille différente, l'un de taille 1 et l'autre de taille n-1.

T(n) = c si n=1
= T(n-1) + n sinon (la fonction Insère est O(n))

Ce qui conduit à $O(n^2)$ (plus complexe que O(nLog(n))

Le tri par fusion subdivise le problème en deux sous problème de taille n/2. Ce qui donne une complexité égale à O(n Log(n)).

2. Le nombre de sous problèmes doit être indépendant de n

La décomposition doit générer un nombre de sous problème constant et indépendant de n. Sinon la solution ne sera pas efficace.
Exemple : Le calcul du déterminant d'une matrice en récursif

Det(M) = M[1,1] Si n=1 (taille de la matrice)
= $\sum_{i=1..n} (-1)^{i+1} * M[1,i] * Det(M(1,j))$ Si n > 1

avec M(i,j) désignant la matrice sans la ligne i et sans la colonne j

Pour une matrice de taille n x n, cette décomposition génère n sous-problèmes (sous-matrices de taille n-1 x n-1)
T(n) = n T(n-1) = n (n-1) T(n-2) = … = n! T(1) --> o(n!)
Alors qu'il existe des algorithmes en $o(n^3)$.