Fiche TD11 : Diviser pour régner

Le paradigme "diviser pour régner"

Le paradigme diviser pour régner (abrégé en DPR, en anglais : divide and conquer) est une technique de conception d'algorithmes reposant sur le principe suivant :

- Diviser le problème en un certain nombre de sous-problèmes de même nature mais de taille moindre.
- Appliquer le même principe aux sous-problèmes de sorte à les résoudre récursivement. Les problèmes de taille suffisamment petite peuvent être résolus directement.
- Combiner les solutions des sous-problèmes afin de produire une solution au problème d'origine.

Nous avons déjà rencontré cette technique à de nombreuses reprises. La conception des algorithmes du tri fusion, du tri rapide, de la recherche dichotomique dans un tableau trié et des algorithmes d'exponentiation (modulaire) rapide par exemple, repose sur cette idée. Notons que dans chacun de ces cas, l'utilisation du paradigme DPR permet d'obtenir une meilleure complexité temporelle qu'avec un algorithme plus naïf. Ce paradigme n'est toutefois pas la panacée, et il est même contre productif dans certains cas. Par exemple, l'utilisation de cette idée pour calculer le n-ème terme de la suite de Fibonacci induit une complexité temporelle et spatiale très importante. De manière générale, lorsque les sous-problèmes sont trop interdépendants, se recoupent, DPR est inadapté, et on préférera utiliser une autre technique : la programmation dynamique.

Le caractère récursif de ce paradigme permet souvent d'exprimer la complexité des algorithmes qui l'utilisent de la manière suivante. Si :

- n est la taille du problème, et on le divise en a sous-problèmes de taille n/b avec $b > 1$ et $a \geq 1$,
- $T(n)$ est le nombre d'opérations effectuées sur une entrée de taille n,
- $D(n)$ est le nombre d'opérations nécessaires pour diviser une entrée de taille n,
- $C(n)$ est le nombre d'opérations nécessaires à la reconstruction de la solution finale à partir des solutions des sous-problèmes,

alors la complexité d'un algorithme DPR suit la relation de récurrence suivante pour n suffisamment grand :

$$T(n) = aT\left(\frac{n}{b}\right) + C(n) + D(n)$$

Par exemple dans le cas de l'exponentiation rapide, $a = 1$, $b = 2$, et $C(n)$ et $D(n)$ sont des $\Theta(1)$. La complexité $T(n)$ de cet algorithme vérifie donc la relation de récurrence $T(0) = \Theta(1)$ et pour tout $n \geq 1$, $T(n) = T(n/2)+\Theta(1)$. Pour analyser la complexité d'un algorithme DPR, c'est-à-dire estimer l'ordre de grandeur de $T(n)$, on utilisera la technique décrite en cours pour analyser le tri fusion : on déduit de la relation de récurrence un arbre dont les noeuds contiennent un nombre d'opérations et tel que la somme des contenus de ces noeuds soit l'ordre de grandeur de $T(n)$.

Exercice 0 *Analyse de relations de récurrence*

Dasn chacun des cas suivants, donner l'ordre de grandeur de la complexité $T(n)$ lorsque cette quantité suit la relation de récurrence proposée. On supposera que $T(n)$ est constant pour $n \leq 1$.

a) $T(n) = T(n/2) + n$. b) $T(n) = 3T(n/3) + n$. c) $T(n) = 4T(n/3) + n$.

d) $T(n) = 2T(n/4) + n^2$. e) $T(n) = 4T(n/2) + \log n$.

Exercice 1 *Fibonacci et DPR ne font pas bon ménage*

On définit la suite de Fibonacci par $F_0 = F_1 = 1$ et $\forall n \geq 2$, $F_n = F_{n-1} + F_{n-2}$.

1. Donner un algorithme récursif naïf permettant de calculer le n-ème terme de cette suite.
2. On note N_n le nombre d'appels à cet algorithme sur l'entier n. Exhiber une relation de récurrence vérifiée par $(N_n)_{n\in\mathbb{N}}$. En déduire que $\forall n \in \mathbb{N}$, $N_n = 2F_n - 1$.
3. En déduire la complexité de cet algorithme.
4. Donner un algorithme de complexité linéaire en n permettant de calculer F_n.
5. a) Rappeler un algorithme permettant de calculer le pgcd de deux nombres a et b.
   b) En admettant que le nombre de divisions euclidiennes dans le calcul du pgcd de a par b est maximal lorsque a et b sont deux termes consécutifs de la suite de Fibonacci, donner l'ordre de grandeur d'un majorant pour le nombre de divisions euclidiennes effectuées lors d'un calcul de pgcd.

Exercice 2 *Sous-tableau maximal*

On considère dans cet exercice des tableaux d'entiers relatifs. Si T est un tableau, un sous-tableau maximal de T est par définition un sous-tableau de cases consécutives de T dont la somme des valeurs est maximale.

1. Exhiber un sous tableau maximal dans le tableau $[7, -3, -5, 2, -1, 8, -9, 3]$.
2. a) Donner un algorithme permettant de calculer un sous-tableau maximal du tableau donné en entrée.
   b) Etudier la complexité de cet algorithme.
3. a) Donner un algorithme linéaire en la taille de l'entrée permettant de calculer un sous-tableau maximal de T si on suppose que ce sous-tableau maximal est situé de part et d'autre du milieu de T.
   b) En déduire comment calculer un sous-tableau maximal de T si on sait calculer un sous tableau-maximal d'un tableau de taille deux fois moindre.
   c) En déduire un algorithme DPR permettant de calculer un sous-tableau maximal de T et étudier sa complexité. Conclure.

Exercice 3 *Nombre d'inversions d'une permutation*

Si $x = [x_1, x_2, ...x_n]$ est un tableau d'entiers, on appelle inversion de x tout couple (i, j) tel que $i < j$ et $x_i > x_j$.

1. Donner le nombre d'inversions dans le tableau $[2, 6, 3, 1, 5]$.
2. a) Proposez un algorithme permettant de calculer le nombre d'inversions d'un tableau d'entiers.
   b) Quelle est la complexité de votre algorithme ?
3. a) A supposer que $x = [x_1, ..., x_{n/2}, ..., x_n]$ et qu'on sache calculer le nombre d'inversions de $[x_1, ..., x_{n/2}]$ et celui de $[x_{n/2+1}, ..., x_n]$, comment peut-on en déduire celui de x ? *Indication : Pour la combinaison, on propose l'idée suivante. On commence par trier les deux sous-tableaux, puis on adapte l'idée du tri fusion pour compter le nombre d'inversions à cheval entre les deux sous-tableaux en temps linéaire.*
   b) En déduire un nouvel algorithme pour calculer le nombre d'inversions d'un tbaleau d'entiers et étudier sa complexité.

Exercice 4 *Recherche d'un élément en deux dimensions*

On considère un tableau M d'entiers à deux dimensions (une matrice) dont chaque ligne et chaque colonne est triée dans l'ordre croissant.

1. a) Ecrire un algorithme naïf prenant en entrée un tel tableau et un entier e et renvoyant vrai si et seulement si e est dans le tableau.
   b) Déterminer la complexité de votre algorithme.
2. a) En s'inspirant de la recherche dichotomique dans un tableau trié, proposer un algorithme DPR résolvant ce problème de recherche. *Indication : Où suffit-il de chercher e si* $e > M[n/2][n/2]$ *?*
   b) Etudier la complexité de ce nouvel algorithme et conclure.

Exercice 5 *Distance minimale dans un nuage de points*

On considère dans cet exercice un nuage $p_1 = (x_1, y_1), ..., p_n = (x_n, y_n)$ de points distincts du plan stockés dans un tableau. On cherche à calculer la distance minimale entre deux points de ce nuage. On suppose qu'on dispose déjà d'un algorithme distance permettant de calculer la distance entre deux points.

1. Donner un algorithme naïf permettant de résoudre le problème de l'énoncé et indiquer sa complexité.

Afin d'améliorer cette complexité, on propose l'idée suivante : on commence par construire deux tableaux P et P', le premier contenant les points classés par ordre croissant des abscisses et le second par ordre croissant des ordonnées. Sans perte de généralité, on peut donc supposer que $P = [p_1, ..., p_n]$ avec $x_1 \leq x_2 \leq ... \leq x_n$.

2. A supposer qu'on connaisse la distance minimale des nuages de points $P_1 = [p_1, ..., p_{n/2}]$ et $P_2 = [p_{n/2}, ..., p_n]$, comment calculer celle de P ? *Indication : Où se trouvent les points qui pourraient éventuellement améliorer les distances minimales de* P_1 *et* P_2 *? Faire un dessin.*
3. Ecrire un algorithme DPR permettant de calculer la distance minimale dans un nuage de points et analyser sa complexité. Est-il problématique de devoir trier le nuage de points en amont ?

Exercice 6 *Multiplication de polynômes*

On considère deux polynômes $P = \sum_{i=0}^{n} a_i X^i$ et $Q = \sum_{j=0}^{n} b_j X^j$ qu'on supposera de même degré pour simplifier dans un premier temps. On représente un polynôme par un tableau indicé à partir de 0 et contenant ses coefficients.

1. Proposer un algorithme permettant de calculer la somme de P et Q et analyser sa complexité.
2. On rappelle que le produit des polynômes P et Q est le polynôme $PQ = \sum_{k=0}^{2n} \left(\sum_{i=0}^{k} a_i b_{k-i} \right) X^k$. Donner un algorithme naïf permettant d'obtenir le produit de deux polynômes.
3. Quelle est la complexité de cet algorithme ?

En notant $M = n/2$ (division entière), on peut toujours écrire P et Q sous la forme

$$P = P_1 + X^M P_2 \quad Q = Q_1 + X^M Q_2$$

avec P_1, P_2, Q_1, Q_2 quatre polynômes de degré inférieur à M.

4. a) Montrer que $PQ = P_1Q_1 + X^M(P_1Q_2 + P_2Q_1) + X^{2M}P_2Q_2$.
   b) En déduire un algorithme DPR pour calculer le produit de deux polynômes et montrer que sa complexité $C(n)$ vérifie $C(n) = 4C(n/2) + \Theta(n)$ pour n suffisamment grand.
   c) En déduire la complexité de ce deuxième algorithme en fonction de n. Commenter.
5. a) Montrer que $PQ = R_1 + X^M(R_2 - R_1 - R_3) + X^{2M}R_3$ avec $R_1 = P_1Q_1$, $R_2 = (P_1 + P_2)(Q_1 + Q_2)$ et $R_3 = P_2Q_2$ et en déduire un nouvel algorithme DPR pour la multiplication de deux polynômes.
   b) Donner une relation de récurrence sur la complexité $C(n)$ de ce nouvel algorithme et en déduire la complexité de ce dernier. Commenter.

Remarque : Il est en fait possible d'obtenir un algorithme de multiplication de polynômes en $O(n \log n)$ en utilisant la transformation de Fourier rapide (qui est encore un algorithme DPR). L'algorithme de la dernière question se nomme l'algorithme de Karatsuba, du nom du mathématicien russe qui le conçut. Cette idée pour calculer un produit de réduire le nombre de multiplications faites, quitte à augmenter le nombre de sommes, se retrouve dans des situations similaires comme en atteste l'exercice suivant.

Exercice 7 *Multiplication matricielle*

On considère deux matrices M et N de taille $n \times n$ qu'on pourra représenter par un tableau de tableaux.

1. Proposer un algorithme permettant de calculer la somme de M et N et analyser sa complexité. Faire de même pour un algorithme permettant de caclculer le produit de M et N. On rappelle que si $MN = P$, le coefficient à la case (i, j) de P est égal à $\sum_{k=1}^{n} m_{i,k} n_{k,j}$ où $m_{i,k}$ désigne le coefficient de M en case (i, k) et $n_{k,j}$ celui de N en case (k, j).
2. Supposons que $n = 2m$. Alors il existe douze matrices de taille $m \times m$ qu'on note $A_1, A_2, B_1, B_2, C_1, C_2, D_1, D_2$, X, Y, Z, T et telles que :

$$M = \begin{pmatrix} A_1 & B_1 \\ C_1 & D_1 \end{pmatrix} \quad \text{et} \quad N = \begin{pmatrix} A_2 & B_2 \\ C_2 & D_2 \end{pmatrix} \quad \text{et} \quad MN = \begin{pmatrix} X & Y \\ Z & T \end{pmatrix}$$

   Montrer que

$$\begin{array}{ll} X = M_1 + M_2 - M_4 + M_6 & \\ Y = M_4 + M_5 & \\ Z = M_6 + M_7 & \text{avec} \\ T = M_2 - M_3 + M_5 - M_7 & \end{array} \quad \begin{array}{ll} M_1 = (B_1 - D_1)(C_2 + D_2) & M_5 = A_1(B_2 - D_2) \\ M_2 = (A_1 + D_1)(A_2 + D_2) & M_6 = D_1(C_2 - A_2) \\ M_3 = (A_1 - C_1)(A_2 + B_2) & M_7 = (C_1 + D_1)A_2 \\ M_4 = (A_1 + B_1)D_2 & \end{array}$$

3. Concevoir un algorithme pour la multiplication matricielle utilisant les égalités précédentes et le paradigme DPR. Etudier sa complexité et comparer avec celle de l'algorithme naïf.

Remarque : Cet algorithme est dû Strassen. La recherche d'algorithmes (et d'implémentations !) efficaces pour la multiplication matricielle est un sujet de recherche actif et important notamment car de nombreux algorithmes d'algèbre linéaire et d'analyse numérique reposent sur cette opération. On donne même un nom au plus petit exposant e tel qu'il existe un algorithme permettant de multiplier deux matrices de taille n en $O(n^e)$: ω. La meilleure borne connue à ce jour est : $\omega < 2.3729$ mais l'algorithme pour lequel cette borne est atteinte est un algorithme galactique.