Leçon 13 : Exemples d'algorithmes utilisant la méthode « diviser pour régner »

I- Présentation

Paradigme 1 : Un algorithme Diviser pour Régner (D&R) s'effectue en 3 étapes :

1 - Division du problème en sous-problèmes indépendants.
2 - Résolution récursive des sous-problèmes
3 - Construction d'une solution du problème global à partir des solutions des sous-problèmes.

Méthode 2 : Calcul de la complexité d'un tel algorithme
Trouver une fonction donnant la taille du problème (comme le nombre d'éléments pour une liste). Définir C la complexité maximale des instances de cette taille, puis trouver une relation de récurrence sur C, pour tenter de la résoudre.

II- Application au calcul formel

II-1) L'exponentiation rapide

Problème 3 : Étant donné un entier a et un entier positif n, calculer a^n.

Solution naïve : n multiplications

Solution D&R :

```
exponentiation_rapide (a, n) :
    Si n = 0 :
        retourner 1                          // étape 1
    x = exponentiation_rapide(a, n/2)        // étape 2
    Si n est pair                            // étape 3
        retourner x * x
    Sinon :
        retourner x * x * a
```

Complexité 4 : Calculons la complexité en nombre de multiplication par rapport à l'entier positif (noté $C(n)$)
Alors $C(n) = C(\lfloor \frac{n}{2} \rfloor) + O(1) = O(\log(n))$

Preuve : 1) C est majoré par $D(n) = D(\lfloor \frac{n}{2} \rfloor) + k$
2) D est croissante
3) $D(2^k)$ est une suite arithmétique donc $D(2^k) = O(k)$
4) $C(n) \overset{1)}{\leq} D(n) \overset{2)}{\leq} D(2^{\lfloor \log(n) \rfloor + 1}) \overset{3)}{=} O(\log n)$

II-2) Multiplication matricielle

Problème 5 : Étant donné $A = (a_{ij})$ et $B = (b_{i,j})$ deux matrices de taille n, on cherche à calculer $A \times B$.

Solution naïve : $O(n^3)$

Solution D&R : (Algorithme de Strassen)

1) Rajouter des 0 pour que A et B soient de tailles paires. Diviser alors A et B en matrices de taille $n/2$.

$$A = \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) \text{ et } B = \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right)$$

2) Calculer récursivement

$\Pi_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22})$
$\Pi_2 = (A_{21} + A_{22}) \times B_{11}$
$\Pi_3 = A_{11} \times (B_{12} - B_{21})$
$\Pi_4 = A_{22} \times (B_{21} - B_{11})$
$\Pi_5 = (A_{11} + A_{12}) \times B_{22}$
$\Pi_6 = (A_{21} - A_{11}) \times (B_{11} + B_{12})$
$\Pi_7 = (A_{12} - A_{22}) \times (B_{21} + B_{22})$

3) Calculer

$$AB = \begin{pmatrix} \Pi_1 + \Pi_4 - \Pi_5 + \Pi_7 & \Pi_3 + \Pi_5 \\ \Pi_2 + \Pi_4 & \Pi_1 - \Pi_2 + \Pi_3 + \Pi_6 \end{pmatrix}$$

Exercice 6 : Prouver la correction de l'algorithme de Strassen

Complexité 7 : $O(n^{\log_2 3})$

Exercice 8 : Prouver cela en reprenant la preuve pour l'exponentiation rapide en considérant $\frac{D(2^k)}{2^k}$ à l'étape 3 de la preuve.

III - Application aux listes

III-1) Recherche dichotomique

Problème 9 : Rechercher un élément a dans une liste L d'éléments triés selon un ordre $\leq$.

Algorithme 10 :

```
recherche_dichotomique (l, a, debut, fin):
    Si fin < debut:
        retourner -1
    m = ⌊(debut + fin)/2⌋
    Si l[m] == a:
        retourner m
    Si l[m] < a:
        retourner recherche_dichotomique (l, a, m+1, fin)
    Sinon:
        retourner recherche_dichotomique (l, a, debut, m-1)
```

Exercice 11 : Ecrire le code en plus de lignes et faire apparaître les 3 étapes de D&R

Complexité 12 : $O(\log(n))$

Exercice 13 : Ecrire une version itérative de cet algorithme

III-2) Tri Fusion

Algorithme 14 :

```
fusion (L1, L2):
    res = []
    i, j = 0
    tant que i < |L1| et j < |L2|:
        Si L1[i] < L2[j]:
            L.ajouter (L1[i])
            i = i+1
        Sinon:
            L.ajouter (L2[j])
            j = j+1
    Ajouter le reste de L1 et de L2 à la fin de res
    retourner res

tri_fusion (L):
    Si L est vide ou contient un seul élément
        Renvoyer L
    L1, L2 = L[:⌊n/2⌋], L[⌊n/2⌋:]      // Etape 1
    A = tri_fusion(L1)                  // Etape 2
    B = tri_fusion(L2)
    Renvoyer fusion(A, B)               // Etape 3
```

Spécification 15 : fusion prend deux listes triés en entrée et renvoie la liste triée contenant les éléments des deux listes.
tri_fusion prend en entrée une liste et renvoie la liste triée

Complexité 16 : $O(n \log(n))$ avec $n = |L|$

Preuve : Exercice

Développement : Correction et terminaison du tri fusion

Application 17 : Si l'on veut chercher si K entiers sont présents parmi N, on peut avec III-1) et III-2) faire cela en $O((N+K)\log N)$ au lieu de $O(K \times N)$.

IV - Application géométrique

IV-1) Plus petite distance dans le plan

Problème 18 : Etant donnés n points $\mathbb{R}^2$, donner la plus petite distance entre deux points

Solution naïve : $O(n^2)$

Solution D&R :

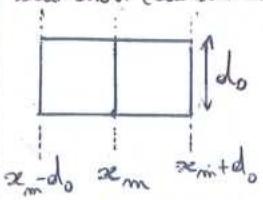
1) Choisir une abscisse x_m séparant les points en deux sous-

ensembles P_1 et P_2 de taille égale (à + ou - 1)

2) Trouver les plus petites distances d_1 et d_2 de P_1 et P_2

3) Rechercher parmi les points suffisament proche de la bande d'abscisse $[x_m - d_0 ; x_m + d_0]$ pour $d_0 = \min(d_1, d_2)$ les deux points les plus proches, séparés de d_3.

L'étape 3) peut se faire en ne regardant que les 7 points suivants (au sens de y)



On a au plus 8 éléments dans ce rectangle, car dans chaque domaine, les points sont au moins à distance d_0, on en a donc au plus 4 points par carré, d'où le resultat en mettant notre point sur le bas du rectangle.

Complexité 19: $O(n(\log n)^2)$

IV-2) Arbre k-dimensionel

Problème 20: Trouver les k plus proches voisins d'un point $y \in \mathbb{R}^k$ parmi un ensemble de n points $x_1, \dots, x_n \in \mathbb{R}^k$

Solution initiale: Stocker nos k valeurs en cours dans une file de priorité et parcourir les n points, en mettant à jour la file de priorité.

Complexité: $O(n \log k)$

Solution D&R: Faire un pré-traitement où l'on stockera nos valeurs dans un arbre binaire de recherche, où l'on partitionnera les données alternativement sur chaque dimension.

On divise ainsi à chaque fois notre espace en deux morceaux. Quand on recherche les plus proches voisins on ne cherche donc que d'un côté, sauf si cela ne suffit pas.

> Développement 2: Présentation de la structure d'arbre k-dimensionnel

Complexité 21: pré-traitement : $O(n \log n)$

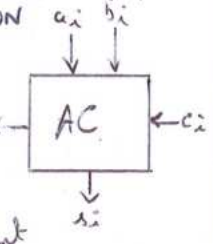
Recherche des k plus proches voisins : $O(\log n \times k)$ en moyenne, $O(n \times k)$ dans le pire des cas.

Remarque 22: On a l'air d'y perdre au change quand, par rapport à la solution naïve. Cette algorithme devient intéressant quand on cherche les k-plus proches voisins de N points. Le pré-traitement étant commun on a en tout du $O(n \log n + N \times k \times \log n)$ au lieu de $O(N * n \times \log k)$.

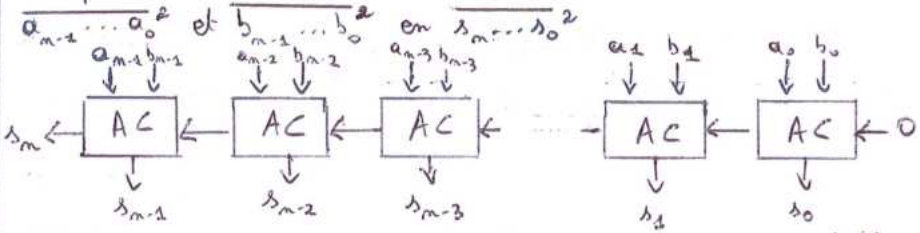
V- Bonus

En architecture des ordinateurs, on peut faire des calculs sur des bits grâce à des portes logiques (ET, OU, XOU, NON, etc.) que l'on peut assembler pour faire des circuits.

On peut créer un additionneur complet effectuant une addition de 3 bits (le résultat (c_{i+1}, s_i) tiendra donc sur deux bits).



On peut donc en déduire un additionneur n bits additionnant $\overline{a_{n-1} \dots a_0}^2$ et $\overline{b_{n-1} \dots b_0}^2$ en $\overline{s_n \dots s_0}^2$



Néanmoins, dans cette solution le chemin critique est en $\Theta(n)$ (le nombre maximal de porte entre une entrée et une sortie), or cela joue sur le délai.

Solution D&R: Un additionneur à retenue anticipée.

Idée: On veut précalculer si les retenues ($les\ c_i$) seront transmises ou générées (suivant les valeurs des a_i et des b_i)

Alors 1) On divise les entrées au milieu (en $\frac{n}{2}$)

2) On regarde si elles génère des retenues, ou si elles les transmettent

3) On en déduit si nos n bits transmettent les retenues ou les transmettent.

On arrive ainsi en profondeur $O(\log n)$ avec $O(n)$ portes à prédire les valeurs des c_i

Ne rien écrire hors du cadre.