IFT436 - Série d'exercices #6 : diviser-pour-régner

Exercice 1: On a vu que l'algorithme de Karatsuba multiplie deux entiers sur n chiffres en séparant le problème en 3 multiplications d'entiers avec $n/2$ chiffres.

a. Avec plus de travail, on peut séparer le problème en 5 multiplications d'entiers avec $n/3$ chiffres chacun (vous n'avez pas à le faire). Que devient la complexité de l'algorithme sous-jacent?

b. Plus généralement, pour tout entier $k \geq 2$, il est possible de séparer le problème en $2k-1$ multiplications d'entiers avec n/k chiffres chacun. Ensuite, on doit faire $O(k)$ additions, chacune en temps linéaire. Montrez que pour toute constante $\alpha > 1$, on peut multiplier en temps $O(n^\alpha)$.

Exercice 2: Soient A et B deux matrices de dimension $n \times n$, où n est une puissance de 2. On veut calculer $A \times B$, c'est-à-dire la multiplication de A et B. L'approche classique prend un temps $O(n^3)$.
L'algorithme de Strassen fait mieux. Nous n'allons pas le détailler ici, mais vous pouvez consulter `https://fr.wikipedia.org/wiki/Algorithme_de_Strassen`.
L'algorithme arrive à séparer le problème en 7 multiplications de matrices de taille $n/2 \times n/2$, en plus de faire $O(1)$ additions et soustractions de matrices, chacune en temps $O(n^2)$.

a. Quelle est la complexité sous-jacente à l'algorithme de Strassen?

b. Le nombre d'additions et soustractions de matrices faites à chaque récursion est au plus 18. Lorsque $n = 1$, aucune addition/soustraction n'est faite.

   Montrez que si n est une puissance de 2, le nombre total d'additions/soustractions faites par l'algorithme est au plus $3 \cdot (7^{\log n} - 1)$

Exercice 3: On nous donne un tableau d'entiers T avec un intervalle $[a \,..\, b] = \{a, a+1, \ldots, b]$ et on veut savoir combien il y a de paire dans T dont la somme est dans $[a \,..\, b]$.
Entrée: tableau d'entiers $T = [t_1, t_2, \ldots, t_n]$ et deux entiers a, b.
Sortie: nombre de paires $\{i, j\}$ avec $i \neq j$ telles que $a \leq t_i + t_j \leq b$.

a. Donnez le pseudo-code d'un algorithme diviser-pour-régner qui implémente la stratégie suivante: on sépare T en deux sous-tableaux de taille $n/2$; on calcule le nombre de paires recherchées récursivement sur les deux moitiés; on trie les deux moitiés de tableau; on calcule le nombre de paires recherchées qui ont un nombre dans la moitié gauche et un nombre dans la moitié droite.

b. Donnez la complexité de l'algorithme de la question précédente.

c. (BONI) Donnez un algorithme diviser-pour-régner en temps $O(n \log n)$ pour ce problème.

Exercice 4: Considérez l'algorithme artificiel suivant:

fonction *algoArtificiel(T) //T est un tableau de taille n*
$A = algoArtificiel(T[1 \,..\, n/5])$
$B = algoArtificiel(T[2n/5 \,..\, 3n/5])$
$traitementQuelconque(A, B)$
$C = algoArtificiel(T[n/5 \,..\, 2n/5])$
$D = algoArtificiel(T[4n/5 \,..\, n])$
$traitementQuelconque(C, D)$
$E = algoArtificiel(T[3n/5 \,..\, 4n/5])$
$F = algoArtificiel(T[1.5n/5 \,..\, 2.5n/5])$
$G = algoArtificiel(T[2.5n/5 \,..\, 3.5n/5])$
return $fonctionQuelconque(A, B, C, D, E, F, G)$

En supposant que la fonction *traitementQuelconque* s'effectue en temps $O(n)$ et $fonctionQuelconque$ en temps $O(1)$, donnez la complexité de cet algorithme.
Et si *traitementQuelconque* s'effectuait en temps $O(n^{1.3})$?

Exercice 5: Considérez le problème du sous-tableau de somme maximum:

Entrée: un tableau $T = [t_1, \ldots, t_n]$ d'entiers positifs ou négatifs.
Sortie: un sous-tableau contigu $T' = [t_i, t_{i+1}, \ldots, t_{i+k}]$ tel que la somme $\sum_{j=i}^{i+k} T[j]$ est maximum parmi tous les choix possibles.

Donnez un algorithme en temps $O(n)$ pour ce problème.
Indice: il y a plusieurs façons d'y arriver. En particulier, il y a une méthode diviser-pour-régner. Pour combiner les sous-solutions, les sous-appels, en plus du sous-tableau maximum, pourraient retourner d'autres informations.

Exercice 6: Dans le problème de la **majorité**, on reçoit un tableau T d'objets quelconques et on veut savoir si un élément est strictement majoritaire. On suppose que $|T| = n$ est une puissance de 2, et on veut savoir s'il existe un élément x qui est présente au moins $n/2 + 1$ fois dans T.

a. Si vous pouvez trier les éléments de T, i.e. $x < y$ est bien défini sur les objets de T, donnez un algorithme en temps $O(n \log n)$ pour le problème de la majorité.

b. Supposons maintenant que $x < y$ n'est pas défini, i.e. vous avez des objets incomparables. Par contre, vous pouvez vérifier l'égalité, donc pour deux éléments de T, vous avez accès à $x = y$ qui retourne $true$ si x et y sont égaux, et $false$ sinon.

Donnez un algorithme en temps $O(n \log n)$ pour le problème de la majorité. Considérez une méthode diviser-pour-régner.

c. (BONI, PLUS DIFFICILE) Donnez un algorithme en temps $O(n)$ qui vérifie si T contient un élément majoritaire. La méthode diviser-pour-régner ne fonctionne peut-être pas ici.

Exercice 7: Vous avez un algorithme récursif dont l'analyse en temps donne la récurrence suivante:

$$f(n) = \begin{cases} 1 & \text{si } n = 0, 1 \\ 2f(n-1) + 8f(n-2) + O(1) & \text{sinon} \end{cases}$$

Donnez la valeur de f en terme de complexité O.

Exercice 8: Soit $G = (V, E)$ un graphe. Un ensemble $C \subseteq V$ est appelé une *clique* si $\forall u, v \in C, uv \in E$. Donc toutes les arêtes possibles entre les éléments de C sont présentes.
Dans le problème de la clique maximum, on cherche la clique $C \subseteq V$ de taille maximum.
Considérez la stratégie suivante: on construit une clique C un sommet à la fois, avec C initialement vide. On prend un sommet $u \in V$, et on branche récursivement sur deux possibilités: soit $u \in C$, ou bien $u \notin C$. On retire ensuite u de G, et on continue récursivement sur les deux cas. Les deux versions de C sont passées récursivement, et quand on atteint un cas terminal, on a une clique qu'on retourne.
Écrivez le pseudo-code qui implémente cette stratégie. Donnez ensuite une récurrence qui exprime le temps requis par cet algorithme, puis évaluez sa complexité.

Exercice 9: Considérez l'algorithme suivant.

fonction *algoBidon*($S = \{s_1, \ldots, s_n\}$) *//S est un ensemble de taille n*
  si $n \leq 10$ **alors**
    return S
  pour $i = 1..8$ **faire**
    $S' = S$ après avoir retiré les éléments s_i, s_{i+1}, s_{i+2}
    $T_i = algoBidon(S')$
  fin
  return $fonctionMagique(T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8)$

Supposez que S' peut être construit en temps $O(1)$, et que $fonctionMagique$ prend un temps $O(1)$.
En terme de O, combien de noeuds l'arbre de récursion de cet algorithme contient-il?