TD 02 – Diviser pour régner (corrigé)

(Maitre) **Exercice 1.** *Suivez le maître !*

Appliquer le *Master Theorem* sur les cas suivants :

1. $T(n) = 9T(n/3) + n$; ☞ Cas 1 : $\Theta(n^2)$.
2. $T(n) = T(2n/3) + 1$; ☞ Cas 2 : $\Theta(\log n)$.
3. $T(n) = 3T(n/4) + n \log n$; ☞ Cas 3 : $\Theta(n \log n)$.
4. $T(n) = 2T(n/2) + n \log n$; ☞ Cas 3 ne marche pas à cause de la régularité ! Exo : $T(n) = \Theta(n \log^2 n)$.
5. $T(n) = 2T(n/2) + n^3$; ☞ Cas 3 : $\Theta(n^3)$.
6. $T(n) = T(9n/10) + n$; ☞ Cas 3 : $\Theta(n)$.
7. $T(n) = 7T(n/3) + n^2$; ☞ Cas 3 : $\Theta(n^2)$.
8. $T(n) = T(n-1) + n$; ☞ Pas de Master Theorem ! A la main : $\Theta(n^2)$.
9. $T(n) = T(\sqrt{n}) + 1$. ☞ $U(m) = T(2^m)$ + Cas 2 : $U(m) = \Theta(\log m)$ donc $T(n) = O(\log \log n)$.

(Meetic) **Exercice 2.** *Mythique*

Un site internet cherche à regrouper ses membres en fonction des goûts musicaux de chacun. Pour cela, chaque membre doit classer par ordre de préférence une liste d'artistes [1]. On dit que deux membres, Arthur et Béatrice, ont des goûts musicaux proches lorsque qu'il y a peu d'*inversions* dans leurs classements : une inversion est une paire d'artiste $\{L, M\}$ telle qu'Arthur préfère L à M et Béatrice préfère M à L. On cherche donc à compter le nombre d'inversion dans les classements d'Arthur et Béatrice.

1. Compter le nombre d'inversion les classements suivants :

   Arthur : Britney Spears, Lady Gaga, Michael Jackson, Madonna, Céline Dion ;

   Béatrice : Lady Gaga, Madonna, Britney Spears, Michael Jackson, Céline Dion.

   ☞ Trois inversions. **Note.** Peut être intéressant de montrer la version graphique.

2. Proposer un algorithme naïf qui résout le problème. Quelle est sa complexité ? ☞ On prend chaque pair d'artistes : $\mathcal{O}(n^2)$.

On cherche maintenant à améliorer l'algorithme précédent en utilisant le paradigme Diviser-Pour-Régner. Pour cela, on coupe le classement de chaque membre en deux sous-classements de même taille, celui des artistes préférés (classement supérieur) et celui des autres artistes (classement inférieur). On compte alors les inversions (L, M) qui peuvent être de deux types : soit L et M apparaissent dans le même sous-classement de Béatrice, soit L et M apparaissent dans deux différents sous-classements de Béatrice (inversions mixtes).

3. On suppose que les deux sous-classements de Béatrice sont triés en fonction du classement d'Arthur [2]. Montrer qu'on peut alors compter les inversions mixtes en temps linéaire. ☞ On numérote les artistes, de sorte qu'Arthur les classe de 1 à n. Le classement de Béatrice est alors un tableau non trié rempli d'éléments de 1 à n. S'il y a une inversion mixte (i, j) avec i dans le classement inférieur et j dans le classement supérieur, alors pour tout $i' \geq i$ dans le classement inférieur, il y a une inversion (i', j). Pour compter en temps linéaire, on procède comme suit :

   On considère les deux sous-classements. Si l'élément le plus petit du classement inférieur est plus petit que celui du classement supérieur, on le supprime. Sinon, on supprime l'autre élément, et on ajoute au compteur d'inversion le nombre d'éléments restants dans le classement inférieur. Temps linéaire immédiat.

4. Donner un algorithme de type Diviser-Pour-Régner qui fonctionne en temps $\mathcal{O}(n \log n)$. ☞ L'idée va être de faire comme avant, mais on doit trier les sous-classements. Notre algo récursif non seulement compte les inversions mais en plus trie le classement : on a un classement, on le coupe en deux, on compte les inversions dans chaque sous-classement et on récupère les deux sous-classement triés. Maintenant, on fusionne les deux sous-classements triés : on utilise la même méthode que dans le cas précédent sauf qu'au lieu de bêtement supprimer les éléments de tête, on crée un nouveau classement. Les éléments arrivent par ordre croissant donc le classement créé est trié. Ainsi, on a compté le nombre d'inversions (qui vaut la somme des nombres d'inversions comptées récursivement et des inversions comptées à cette étape) et on a retourné un classement trié.

   La complexité : $T(n) = 2T(\lceil n/2 \rceil) + \mathcal{O}(n)$. Donc $T(n) = \mathcal{O}(n \log n)$.

1. Le classement est un ordre total.
2. Si L et M apparaissent dans le même sous-classement de Béatrice, alors ils apparaissent dans le même ordre que dans le classement d'Arthur.

(pgdpg) **Exercice 3.** *Deuxième plus grand élément*

On s'intéresse dans cet exercice à la **complexité dans le pire des cas et en nombre de comparaisons** des algorithmes.

1. Pour rechercher le plus grand et deuxième plus grand élément de n entiers, donner un algorithme naïf et sa complexité.

   ☞ Algorithme naïf : recherche du premier maximum, puis du second.

---

début
  $max_1 \leftarrow T[1]$;
  pour i **allant de** 2 **à** n **faire**
    si $T[i] > max_1$ **alors**
      $max_1 \leftarrow T[i]$;
      $posmax_1 \leftarrow i$;
  si $posmax_1 \neq 1$ **alors** $max_2 \leftarrow T[1]$ **sinon** $max_2 \leftarrow T[2]$;
  pour i **allant de** 2 **à** n **avec** $i \neq posmax_1$ **faire**
    si $T[i] > max_2$ **alors**
      $max_2 \leftarrow T[i]$;
  retourner max_1, max_2;

Nombre de comparaisons :

Premier maximum, sur n valeurs :	$n-1$
Second maximum, sur n - 1 valeurs :	$n-2$
	$2n-3$

2. Pour améliorer les performances, on se propose d'envisager la solution consistant à calculer le maximum suivant le principe d'un *tournoi* (tournoi de tennis par exemple). Plaçons-nous d'abord dans le cas où il y a $n = 2^k$ nombres qui s'affrontent dans le tournoi. Comment retrouve-t-on, une fois le tournoi terminé, le deuxième plus grand ? Quelle est la complexité de l'algorithme ? Dans le cas général, comment adapter la méthode pour traiter n quelconque ?

   ☞ Cas où $n = 2^k$:

   On calcule le premier maximum à la façon d'un tournoi de tennis. On cherche ensuite le second maximum, en prenant le maximum parmi les adversaires que le premier à rencontré. Ainsi, dans l'arbre donné à la figure 1, le second maximum est nécessairement un élément contenu dans le chemin rouge.

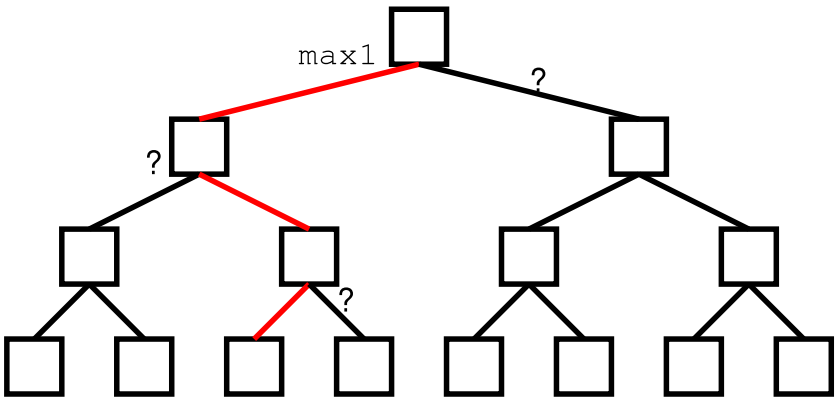

FIGURE 1 – Arbre des «compétitions» effectuées

Nombre de comparaisons :

Premier maximum :	$n-1 = 2^k - 1$
Second maximum :	$k-1$
	$2^k + k - 2 = n + log_2 n - 2$

Cas général : n quelconque

On cherche k tel que $2^k \geq n$: prendre $\min\{k | 2^k \geq n\} = \lceil \log_2 n \rceil$ (voir Figure 2). Avec le raisonnement d'avant on a $n-1$ matches pour trouver le premier maximum, et on a des branches de longueur au plus égale à $\lceil \log_2 n \rceil$, donc dans le pire des cas on a à faire $\lceil \log_2 n \rceil - 1$ comparaisons entre les noeuds battus par le max. Soit un total de $n + \lceil \log_2 n \rceil - 2$.

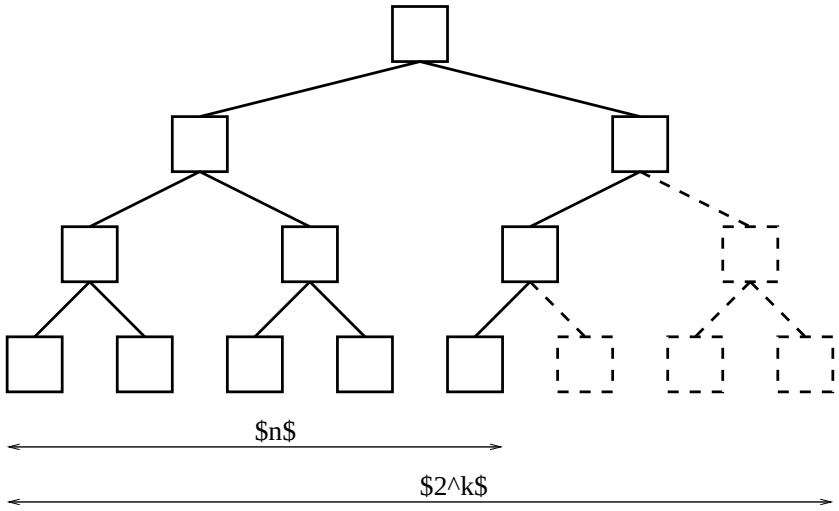

FIGURE 2 – Arbre lorsque n est quelconque

Au pire, nombre de comparaisons :
Premier maximum : $n-1$
Second maximum : $\lceil log_2 n \rceil - 1$
$n + \lceil log_2 n \rceil - 2$

Remarque : plus formellement pour évaluer la hauteur maximale de l'arbre représentant le tournoi, faire une récurrence sur $\forall d \geq 0, \forall n \geq 2\ 2^d < n \leq 2^{d+1} \Rightarrow h(n) = d+1$.

3. Montrons l'optimalité de cet algorithme en fournissant une borne inférieure sur le nombre de comparaisons à effectuer. Nous utiliserons la méthode des *arbres de décision*.

☞ **Remarque : Arbre de décision d'un algorithme :** arbre représentant toutes les exécutions possibles de l'algorithme sur toutes les donnés d'une certaine taille :
— feuilles : résultats des différentes exécutions (plusieurs peuvent donner le même résultat)
— nœuds internes : tests pour aiguillage, ici nœud = comparaison (gauche : oui, droite : non) on a donc un arbre binaire.

☞ **Remarque : Complexité en tests :** une exécution = une branche, donc le nombre de comparaisons est égal à la hauteur de la branche, soit la hauteur de l'arbre.

(a) Montrer que tout arbre de décision qui calcule le maximum de N entiers a au moins 2^{N-1} feuilles.

☞ Pour chercher le maximum parmis N valeurs, on doit effectuer $N-1$ comparaisons.

On a donc 2^{N-1} feuilles dans l'arbre de décision (nombre de feuilles d'un arbre binaire de hauteur $N-1$).

(b) Montrer que tout arbre binaire de hauteur h et avec f feuilles vérifie $2^h \geq f$.

☞ Par récurrence sur la hauteur h de l'arbre
— Pour $h = 0$: 1 feuille au plus
— On considère vrai jusqu'à la hauteur h : $2^h \geq f$
— On considère un arbre de hauteur $h+1$, on a alors deux cas, soit la racine a un seul fils soit il en a deux.
  — un fils : on a alors le nombre de feuilles qui correspond à celui de l'arbre partant du fils, qui est de hauteur h , et $2^{h+1} \geq 2^h$ ce qui va bien.
  — deux fils : on a alors le nombre de feuilles qui correspond à la somme de celles des deux sous arbres partants des deux fils : $f = f_1 + f_2$, en ayant pour chacun une hauteur maximale de h soit : $2^{h+1} \geq 2^h + 2^h \geq 2^{h_1} + 2^{h_2} \geq f_1 + f_2 \geq f$ cqfd.

(c) Soit A un arbre de décision résolvant le problème du plus grand et deuxième plus grand de n entiers, minorer son nombre de feuilles. En déduire une borne inférieure sur le nombre de comparaisons à effectuer.

☞ La Figure 3 présente un arbre de décision pour le maximum et deuxième maximum de 4 valeurs. On partitionne les feuilles de A selon la valeur du premier maximum pour former les A_i : A_i est au final l'arbre A dont on a enlevé tous ce qui n'aboutissait pas à une feuille concluant que le premier maximum était i. On supprime alors les nœuds où il y a un test avec $T[i]$, ces nœuds sont forcément nœud avec un seul fils (sinon ce serait une feuille). Ces A_i sont des arbres donnant un maximum parmi n-1 éléments donc ils ont chacun un nombre de feuilles tel que : nombre de feuilles de $A_i \geq 2^{n-2}$ Donc en considérant A comme la "fusion" de ces arbres qui en forme une partition, on a :

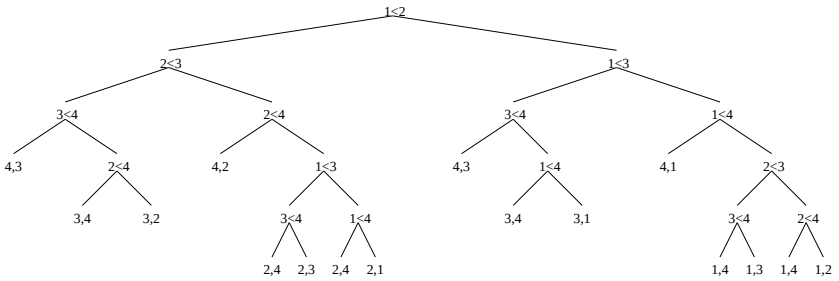

FIGURE 3 – Arbre de décision pour 4 valeurs : les feuilles donnent le max et 2^{me} max

$$\begin{aligned}
\textit{nombre de feuilles de } A &= \sum_{i=1}^{n} \textit{nombre de feuilles de } A_i \\
&\geq \sum_{i=1}^{n} 2^{n-2} \\
&\geq n2^{n-2} \\
2^{\textit{hauteur}} &\geq n2^{n-2} \\
\textit{hauteur} &\geq \lceil \log_2(n2^{n-2}) \rceil \\
&\geq n - 2 + \lceil \log_2 n \rceil
\end{aligned}$$

(MultiplicationEntiers) **Exercice 4.** *Reconstruisons les tables de multiplication*

Soient deux entiers x et y codés sur n bits (on supposera que n est une puissance de 2). On souhaite multiplier ces deux entiers entre eux, en travaillant au niveau des bits. La multiplication de deux entiers de n bits se fait de manière triviale en $\mathcal{O}(n^2)$, la multiplication d'un entier par une puissance de 2, et l'addition de 2 entiers se font en temps linéaire $\mathcal{O}(n)$.

1. La méthode diviser pour régner n'est pas toujours meilleure qu'un algorithme naïf. Pour illustrer cela, donnez un algorithme diviser pour régner ayant une complexité en $\mathcal{O}(n^2)$ pour multiplier deux entiers x et y de n bits chacun.

   ☞

   On peut écrire x et y de la façon suivante :

   $$\begin{aligned} x &= \boxed{x_L}\boxed{x_R} = 2^{n/2}x_L + x_R \\ y &= \boxed{y_L}\boxed{y_R} = 2^{n/2}y_L + y_R \end{aligned}$$

   On peut alors multiplier x et y de la façon suivante :

   $$xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R$$

   On a donc besoin de 4 appels récursifs pour multiplier des entiers de taille $n/2$ pour $x_L y_L$, $x_L y_R$, $x_R y_L$ et $x_R y_R$. Il faut ensuite faire les multiplications par $2^{n/2}$ et les additions, cela peut être fait en $\mathcal{O}(n)$. On a donc la formule de récurrence suivante :

   $$T(n) = 4T(n/2) + \mathcal{O}(n)$$

   ce qui nous donne une complexité totale en $\mathcal{O}(n^2)$ par le *master theorem*.

2. Proposez un algorithme diviser pour régner ayant une complexité inférieure à $\mathcal{O}(n^2)$. Donnez la formule de récurrence pour votre algorithme et sa complexité finale (en $\mathcal{O}$). On supposera pour simplifier que l'addition/multiplication de deux nombres de taille n est un nombre de taille n.

   ☞

   En réorganisant un peu l'expression précédente, on peut améliorer l'algorithme pour avoir une complexité en $\mathcal{O}(n^{\log_2 3})$. Seulement trois multiplications sont suffisantes : $x_L y_L$, $x_R y_R$ et $(x_L + x_R)(y_L + y_R)$, puisque $(x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R = x_L y_R + x_R y_L$.

On a donc l'algorithme suivant :

Algorithm 1: multiplier(x,y)

début
  Data: 2 entiers x et y de n bits
  si $n = 1$ **alors**
    retourner xy
  $p_1 = multiplier(x_L, y_L)$;
  $p_2 = multiplier(x_R, y_R)$;
  $p_3 = multiplier(x_L + y_R, y_L + y_R)$;
  retourner $p_1 \times 2^n + (p_3 - p_1 - p2) \times 2^{n/2} + p_2$

On a donc la formule de récurrence suivante :

$$T(n) = 3T(n/2) + \mathcal{O}(n)$$

ce qui nous donne $\mathcal{O}(n^{\log_2 3}) \approx \mathcal{O}(n^{1.59})$.