Corrigé TD n° 8 : Diviser pour régner

EXERCICE 1

Minimum d'un tableau : On utilise une fonction auxiliaire qui prend en plus en argument des aindicies de début et de fin de traitement de tableau pour éviter les recopiage et l'explosion de la complexité spatiale.

OCAML
```
let minimum tableau =
  (* 'a Array -> 'a *)
  let rec min_dpr deb fin =
    (* int -> int -> 'a *)
    if fin = deb then
      tableau.(deb)
    else
      let milieu = (deb + fin) / 2 in
      min (min_dpr deb milieu) (min_dpr (milieu + 1) fin)
  in
  let n = Array.length tableau in
  if n = 0 then
    failwith "tableau vide !"
  else
    min_dpr 0 (Array.length tableau - 1)
;;
```

On obtient une équation de complexité $T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \Theta(1)$, soit pour une puissance de 2, $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(1)$ ce qui conduit avec les techniques habituelles (suite ou arbre) à une complexité de $\Theta(n)$. On ne gagne rien par rapport à un parcours linéaire du tableau.

Somme des termes d'une liste d'entiers : Comme pour le tri fusion, on découpe la liste en deux sous-listes de taille (presque-)égales.

OCAML
```
let rec division liste =
  (* 'a list -> 'a list * 'a list *)
  match liste with
  | [] -> [], []
  | tete :: [] -> liste, []
  | tete1 :: tete2 :: queue ->
     let liste1, liste2 = division queue in
     tete1 :: liste1, tete2 :: liste2
;;

let rec somme_liste liste =
  (* int list -> int *)
  match liste with
  | [] -> 0
  | tete :: [] -> tete
  | _ ->
     let liste1, liste2 = division liste in
     somme_liste liste1 + somme_liste liste2
;;
```

Comme pour le tri fusion, la division a un coût linéaire mais ici, il n'y a pas besoin de fusion.
On obtient une équation de complexité $T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \Theta(n)$, soit pour une puissance de 2, $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$ ce qui conduit avec les techniques habituelles (suite ou arbre) à une complexité de $\Theta(n \log n)$: pas intéressant.
Remarquons qu'en travaillant sur un tableau comme dans l'exemple précédent, la fusion aurait été à coût constant et on aurait retrouvé le $\Theta(n)$ de la méthode standard.

Exponentiation rapide en décomposant l'exposant modulo 3 :

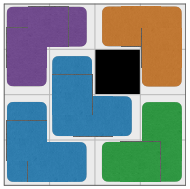
OCaml
```
let rec expo_modulo_3 x puissance =
  if puissance = 0 then
    1.
  else
    let reste = puissance mod 3 in
    let x_cube = x *. x *. x in
    let y = expo_modulo_3 x_cube (puissance / 3) in
    match reste with
    | 0 -> y
    | 1 -> x *. y
    | _ -> x *. x *. y
;;
```

Notons $T(n)$ le coût pour effectuer exponentiation rapide en décomposant l'exposant n modulo 3.
On a l'équation $T(n) = T\left(\left\lfloor \frac{n}{3} \right\rfloor\right) + \Theta(1)$ ce qui conduit avec les techniques habituelles (suite ou arbre) à une complexité de $T(n) = \Theta(\log_3 n) = \Theta(\log n)$.

La seule chose qu'on améliore dans la complexité de cette exponentiation trinaire est la constante du Θ par rapport à l'exponentiation binaire.

Exercice 2

1. On a toujours $3 \times k \neq 2 \times 2^k = 2^{k+1}$
2. Une solution possible est



3. Soit $k \in \mathbb{N}$ tel que $k \geq 1$.

On note : $[P_k]$ l'assertion « On peut recouvrir un carré de côté 2^k laissant vide une case arbitraire que l'on appelle aussi case inutilisable. »

- Les assertions $[P_1]$ et $[P_2]$ sont vérifiées.
- On suppose $[P_k]$ vérifiée pour $k \in \mathbb{N}$ fixé.

  On dispose d'un "grand" carré C_0 de côté 2^{k+1} laissant vide une case arbitraire c. On partitionne par le milieu le carré C_0 en

  ⋆ Un carré C_1 de côté 2^k contenant c.

  ⋆ Trois carrés C_2, C_3, C_4 chacun de côté 2^k (formant un motif.)

  Au centre du "grand" carré C_0, on dispose un motif touchant C_2, C_3, C_4.

  C_2, C_3, C_4 sont ainsi devenus des carrés de côté 2^k ayant chacun une case inutilisable. Avec l'hypothèse de récurrence, on peut paver d'une part C_1 et d'autre part C_2, C_3, C_4.
  On a ainsi recouvert C_0.

L'assertion $[P_k]$ est donc démontrée par récurrence.

Exercice 3

1. La complexité de l'algorithme naïf permettant de calculer la somme de deux polynômes de degré n est en $\Theta(n)$.
2. Avec $AB = \sum_{k=0}^{2n} c_k X^k$ On a $c_k = \sum_{i=0}^{k} a_i b_{k-i}$
   Soit un nombre d'additions

$$\sum_{k=0}^{2n} k = \frac{2n(2n+1)}{2} = 2n^2 + n$$

   Soit un nombre de multiplications

$$\sum_{k=0}^{2n} \sum_{i=0}^{k} 1 = \sum_{k=0}^{2n} (k+1) = \frac{(2n+1)^2}{2}$$

La complexité de l'algorithme naïf permettant de calculer le produit de deux polynômes de degré n est en $\Theta\left(n^2\right)$.

3. $n = 2^p$, $m = \frac{n}{2} = 2^{p-1}$

$$P = X^m P_1 + P_2 \text{ et } Q = X^m Q_1 + Q_2$$

En développant

$$PQ = (X^m P_1 + P_2)(X^m Q_1 + Q_2) = X^{2m} P_1 Q_1 + X^m (P_1 Q_2 + P_2 Q_1) + Q_1 Q_2$$

Notons $C(n)$ le coût du calcul de PQ avec $\deg P = n = 2^p$.
Puisqu'on accède aux coefficients de chaque polynôme avec un coût constant, il n'y a pas de coût de décomposition. 4 appels récursifs sont nécessaires, et la recomposition nécessite 3 additions de coût linéaire. Le nombre de multiplications vérifie donc la relation

$$C(n) = 4C\left(\frac{n}{2}\right) + \Theta(n)$$

Avec le `Master Theorem` (dont on retrouve l'énoncé à l'aide d'un arbre !)
Dans le cas présent $a, d, k = 4, 1, 2$
Il vient : $k^d = 2^1 = 2 < 4 = a$.
Ce qui fournit $C(n) = \Theta(n^{\log_k a}) = \Theta(n^{\log_2 4}) = \Theta(n^2)$

4. On a

$$P_1 Q_2 + P_2 Q_1 = (P_1 + P_2)(Q_1 + Q_2) - (P_1 Q_1 + P_2 Q_2)$$

Posons

- $R_1 = P_1 Q_1$
- $R_2 = (P_1 + P_2)(Q_1 + Q_2)$
- $R_3 = P_2 Q_2$

Il n'est plus nécessaire que d'utiliser 3 appels récursifs. La relation de récurrence prend cette fois la forme $C(n) = 3C\left(\frac{n}{2}\right) + \Theta(n)$.
Avec le `Master Theorem` (dont on retrouve l'énoncé à l'aide d'un arbre !)
Dans le cas présent $a, d, k = 3, 1, 2$.
Il vient $k^d = 2^1 = 2 < 3 = a$.
Ce qui désormais conduit à $C(n) = \Theta\left(n^{\log_k a}\right) = \Theta\left(n^{\log_2 3}\right)$ et $\log_2 3 \approx 1,585$.

EXERCICE 4

Soit $n \in \mathbb{N}^*$. Intéressons-nous au produit matriciel.

1. (a) Si M_1 et M_2 sont deux matrices de tailles $n \times n$,
l'addition matricielle nécessite n^2 additions scalaires.

(b) Si M_1 et M_2 sont deux matrices de tailles $n \times n$,
le produit matriciel nécessite $n^2(n-1)$ d'additions scalaires et
n^3 multiplications scalaires.

2. (a) On effectue le produit par blocs :

$$M_1 M_2 = \begin{pmatrix} A_1 A_2 + B_1 C_2 & A_1 B_2 + B_1 D_2 \\ C_1 A_2 + D_1 C_2 & C_1 B_2 + D_1 D_2 \end{pmatrix}$$

Cette méthode nécessite a priori 8 multiplications et 4 additions sur des matrices $m \times m$.

(b) Les valeurs de de a , k et d sont $a = 8, k = 2, d = 2$.

(c) En utilisant le Master Theorem, comme $a = 8, k = 2, d = 2$, on peut estimer le coût c_n $k^d = 2^2 = 4 < 8 = a$ et il vient $c_n = \Theta\left(n^{\log_k a}\right) = \Theta\left(n^{\log_2 8}\right) = \Theta\left(n^3\right)$.
On constate que l'on ne gagne donc rien par rapport à l'algorithme naïf.

3. (a) Les valeurs de de a , k et d sont $a = 7, k = 2, d = 2$.

(b) En utilisant le Master Theorem, comme $a = 7, k = 2, d = 2$, on peut estimer le coût c_n.
$k^d = 2^2 = 4 < 7 = a$ et il vient $c_n = \Theta\left(n^{\log_k a}\right) = \Theta\left(n^{\log_2 7}\right) = \Theta\left(n^{2,807}\right)$
On constate désormais un certain gain par rapport à la méthode naïve.