TD 9 - Diviser pour régner

1 Les incontournables

Exercice 1 [**Tri par partition fusion d'une liste**]

1. Écrire une fonction `partition : 'a list -> 'a list * 'a list` qui coupe une liste en 2 (l'ordre des éléments n'est pas important)
2. Écrire une fonction `fusion : 'a list -> 'a list -> 'a list` qui recevant 2 listes l_1 et l_2 ordonnées par ordre croissant renvoie la liste ordonnée contenant tous les éléments de l_1 et de l_2.
3. En déduire une fonction `tri_partition_fusion : 'a list -> 'a list` qui trie une liste selon le procédé de partition-fusion.

Exercice 2 La *suite de Fibonacci* est définie par les relations :

$$F_0 = 1,\ F_1 = 1 \quad et \quad \forall n \geqslant 1,\ F_{n+1} = F_n + F_{n-1}$$

1. Écrire une fonction récursive `fibo_naif` utilisant cette définition telle que `fibo_naif n` renvoie F_n, et rappeler pourquoi cet algorithme est inefficace.
2. On note $u_n = F_n$ et $v_n = F_{n+1}$. Remarquer que les deux suites (u_n) et (v_n) vérifient les relations de récurrence :
$$u_0 = 1,\ v_0 = 1 \text{ et } \forall n \geqslant 0,\ \begin{cases} u_{n+1} = v_n \\ v_{n+1} = u_n + v_n \end{cases}$$
Écrire une fonction `fibo_suites` utilisant cette nouvelle relation. Quelle est la complexité de ce programme (en terme de nombre d'appels) ?
3. On cherche à déterminer un algorithme du type "diviser pour regner" pour calculer les termes de cette suite.
   a. Démontrer la relation : $\forall n, p \geqslant 1,\ F_{n+p} = F_nF_p + F_{n-1}F_{p-1}$.
   b. En déduire une expression de F_{2n} et F_{2n+1} en fonction de F_n et F_{n-1}.
   c. Ecrire une fonction `fibo_DpR` qui calcule F_n selon la méthode diviser pour régner
   d. Estimer sa complexité.
4. On souhaite conserver les temps d'exécution de ces trois fonctions en fonction de l'indice du terme F_n demandé. A l'aide de la fonction `time` du module `Sys`, écrire une fonction `temps : (int -> int) -> int -> float array` telle que `temps f n` renvoie un tableau indiquant les temps de calcul pour pour les termes $F_0, ...F_n$ avec la fonction `f`.
5. On cherche stocker dans un fichier CSV les temps de calculs de F_n en fonction de n avec ses 3 méthodes pour obtenir les courbes avec un tableur. (Pensez-y pour vos TIPE !). Un fichier CSV est un fichier texte où les colonnes de données sont séparées par des points virgules. Sous OCaml, voici 3 fonctions pour manipuler sur les fichiers
   - `open_out : string -> out_channel` : ouvre un fichier en écriture
   - `output_string : out_channel -> string -> unit` : écrit une chaîne de caractères dans un fichier.
   - `close_out : out_channel -> unit` : ferme un fichier préalablement ouvert en écriture.

   Voici un exemple de programme qui écrit bonjour dans le fichier `coucou.txt` placé dans le répertoire courant
```
let ecrit_bonjour fichier =
        let oc = open_out fichier in
                output_string oc "bonjour";
                close_out oc
;;
ecrit_bonjour "H:/coucou.txt");;
```
6. Afficher alors les courbes du temps à l'aide d'un tableur.

2 Pour s'entrainer

Exercice 3 Recherche du maximum dans un tableau...

1. Écrire une fonction `max_i` itérative qui retourne le maximum d'un tableau d'entiers. Combien de comparaison d'éléments sont réalisés ?
2. On propose cette fonction récursive `max_r` basée sur le principe « diviser pour régner ».

```
let max_r t =
    let rec maxR deb fin =
        if deb = fin
        then t.(deb)
        else let m = (deb+fin)/2 in
            max (maxR deb m) (maxR (m+1) fin)
    in maxR 0 (Array.length t - 1)
;;
```

   Combien d'appels à la fonction `max` sont effectués ? Que dire de cette nouvelle solution ?

Exercice 4 En partant du constat : $a^n = \begin{cases} \left(a^{\frac{n}{2}}\right)^2 \text{ si } n \text{ pair} \\ \\ a \times \left(a^{\frac{n-1}{2}}\right)^2 \text{ sinon} \end{cases}$

1. Écrire une fonction `puiss` de type `int -> int > int` qui recevant 2 arguments a et n, renvoie a^n.
2. Adapter votre programme pour qu'il affiche le résultat modulo p. Et vérifier (à la main et avec votre programme) que $2023^{2023} \equiv 5[13]$

Exercice 5 Dans cet exercice, on considère une matrice d'entiers M telle que chaque ligne et chaque colonne soit rangée par ordre croissant. Voici un exemple d'un tel tableau, à 4 lignes et 5 colonnes :

$$\begin{pmatrix} 2 & 14 & 25 & 30 & 69 \\ 3 & 15 & 28 & 30 & 81 \\ 7 & 15 & 32 & 43 & 100 \\ 20 & 28 & 36 & 58 & 101 \end{pmatrix}.$$

Le but de l'exercice est de rechercher efficacement un élément v dans un tel tableau. Pour simplifier, on supposera que ce tableau comporte $n = 2^k$ lignes et colonnes numérotées de 0 à $n-1$.

1. Écrire une fonction `cree` telle que `cree k` permette de créer un tel tableau aléatoire de taille 2^k. On pourra aller chercher dans le module `Random` comment générer des nombres aléatoires.
2. On distingue les deux valeurs $x = M\left(\frac{n}{2} - 1; \frac{n}{2} - 1\right)$ et $y = M\left(\frac{n}{2}; \frac{n}{2}\right)$. Montrer que si $v > x$, on peut éliminer une partie (à préciser) du tableau pour poursuivre la recherche. Préciser ce qu'il est possible de faire lorsque $v < y$.
3. En déduire une méthode « diviser pour régner » pour résoudre ce problème, et préciser le coût dans le pire des cas, en nombre de comparaisons, de cette méthode.
4. Programmer la méthode.

Correction 1

1. Une solution (Pierre THOREZ) avec 2 accumulateurs et un paramètre booléen g indiquant si on place dans la première ou seconde liste :

```
let partition l=
        let rec partitionR l l1 l2 g =
                match l with
                        |[] -> l1, l2
                        |h::t when g -> (partitionR t (h::l1) l2 false)
                        |h::t -> (partitionR t l1 (h::l2) true)
        in partitionR l [] [] false
;;
```

une autre (par Mathilde LECAT) qui nécessite un accumulateur de moins (mais de connaitre la longueur de la liste) :

```
let partition l =
        let n=(List.length l -1)/2 in
        let rec partitionR g d c=
                match c with
                  | c when c>n -> g , d
                  |_ -> partitionR ((List.hd d)::g) (List.tl d) (c+1)
                in partitionR [] l 0 ;;
```

2. Pour la fusion :

```
let rec fusion l1 l2 =
        match (l1, l2) with
          | [], l2 -> l2
          | l1, [] -> l1
          | t1::q1, (t2::q2 as l2) when t1 < t2 -> t1::fusion q1 l2
          | l1, t2::q2 -> t2::fusion l1 q2
;;
```

3. Reste à appliquer l'algorithme :

```
let rec tri_partition_fusion = function
        | [] -> []
        | [x] -> [x]
        | l -> let l1, l2 = partition l in
                fusion (tri_partition_fusion l1) (tri_partition_fusion l2)
;;
```

Correction 2

Correction 3 1. La première solution ne devrait pas poser de difficulté :

```
let max_i t =
        let m = ref t.(0) in
        for i = 1 to Array.length t - 1 do
                if !m < t.(i) then m := t.(i)
        done;
        !m
;;
```

2. Si on note n la taille du tableau et T le nombre d'appel à la fonction `max`, on a $T(1) = 0$ et $T(n) = 1 + 2T(n/2)$ si n est pair. En posant $u_n = T(2^n)$, on obtient une suite arithmético-géométrique que l'on résout. On trouve finalement $T(n) = n - 1$

Correction 4

```
let rec puiss a = function
        | 0 -> 1
        | n when n mod 2 = 0 -> puiss (a*a) (n/2)
        | n -> a * puiss (a*a) (n/2)
;;

let puissm a n p =
        puissmR (a mod p) p n where
                rec puissmR a p = function
                        | 0 -> 1
                        | n when n mod 2 = 0 -> puissmR (a*a mod p) p (n/2)
                        | n -> (a * puissmR (a*a mod p) p (n/2)) mod p
;;
```

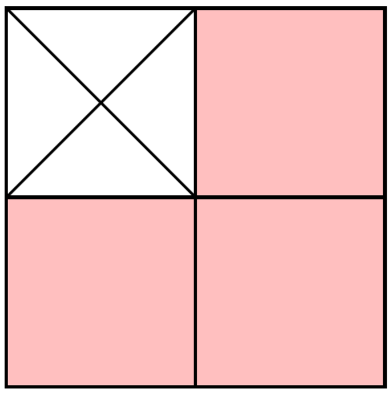
Correction 5

1. Une proposition (il y en a d'autres !) :

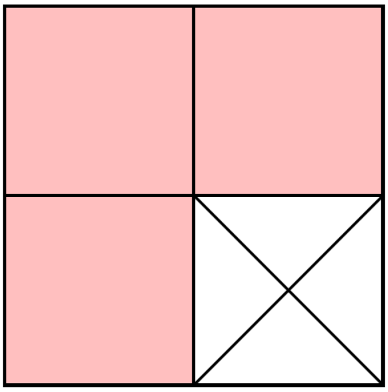
```
let rec puiss2 = function (* Calcul de 2^n *)
        | 0 -> 1
        | k -> 2*puiss2 (k-1)
;;

let cree k =
   let n = puiss2 k in
      let m = Array.make_matrix n n 0 in
         m.(0).(0) <- Random.int 9;
         for i = 1 to n-1 do
            m.(0).(i) <- m.(0).(i-1) + Random.int 5;
         done;
         for i = 1 to n-1 do
            m.(i).(0) <- m.(i-1).(0) + Random.int 5;
         done;
         for i = 1 to n-1 do
            for j = 1 to n-1 do
               m.(i).(j) <- max m.(i-1).(j) m.(i).(j-1) + Random.int 5
            done;
         done;
         m;
;;
```

2. Si $z = b(i_0, j_0)$, il est facile de constater que pour tout $i \leqslant i_0$ et $j \leqslant j_0$ on a $M(i, j) \leqslant z$. À l'inverse, pour tout $i > i_0$ et $j > j_0$ on a $b(i, j) > z$. Ainsi, si $v > x$ on peut chercher v dans la partie du tableau correspondant aux indices $i \geqslant n/2$ et $j \geqslant n/2$.



À l'inverse, si $v < y$ on peut chercher v dans la partie du tableau correspondant aux indices $i < \dfrac{n}{2}$

et $j < \dfrac{n}{2}$. 

3. Sachant que $x \leqslant y$, l'une de ces deux conditions est forcément vérifiée (ou alors c'est qu'on a trouvé v). Dans le pire des cas, après ces deux tests la recherche se ramène à trois tableaux de tailles $\dfrac{n}{2} \times \dfrac{n}{2}$. Le coût dans le pire des cas vérifie donc la relation : $c(n) = 3c(n/2) + 1$. Il s'agit d'un coût en $O(n^{\log_2 3})$.
4. La fonction `recherche`

```
let recherche v m =
        let rec rechercheR l0 c0 = function
                | 1 -> m.(l0).(c0) = v
                | d when v > m.(l0+d/2-1).(c0+d/2-1)
                               -> rechercheR l0 (c0+d/2) (d/2)
                               || rechercheR (l0+d/2) c0 (d/2)
                               || rechercheR (l0+d/2) (c0+d/2) (d/2)
                | d -> rechercheR l0 (c0+d/2) (d/2)
                               || rechercheR (l0+d/2) c0 (d/2)
                               || rechercheR l0 c0 (d/2)
        in rechercheR 0 0 (Array.length m)
;;
```