DS 3 du 2 juin - 2 heures

Calculatrice, téléphone et documents interdits

Les programmes non commentés ou dont le fonctionnement n'est pas expliqué ne seront pas relus. Une présentation raisonnablement aérée (avec une identation convenable des structures de choix ou de répétition), ainsi que des résultats soulignés ou encadrés, sont des éléments de l'évaluation... C'est vous qui voyez...

Exercice 1 [**Récompense pour les braves !**]

1. Écrire une fonction récursive `colle : 'a list -> 'a list list -> 'a list list` qui associe à une liste d'éléments (l'alphabet) et à une liste de listes constituées d'éléments de l'alphabet, la liste de listes obtenues en ajoutant en tête l'un des éléments de l'alphabet. On souhaite obtenir par exemple

```
# colle [1;2;3;4] [[5;6];[7];[]];;
- : int list list = [[1; 5; 6]; [1; 7]; [1];
[2; 5; 6]; [2; 7]; [2]; [3; 5; 6]; [3; 7]; [3];
[4; 5; 6]; [4; 7]; [4]]
```

2. Écrire une fonction récursive `mots : 'a list -> int -> 'a list list` qui à une liste `alphabet` et un entier `n`, associe la liste de listes correspondant à tous les mots de longueur `n` construits avec les éléments de l'alphabet. On souhaite obtenir par exemple :

```
mots [1;2] 4;;
- : int list list = [[1; 1; 1; 1]; [1; 1; 1; 2];
       [1; 1; 2; 1]; [1; 1; 2; 2]; [1; 2; 1; 1];
       [1; 2; 1; 2]; [1; 2; 2; 1]; [1; 2; 2; 2];
       [2; 1; 1; 1]; [2; 1; 1; 2]; [2; 1; 2; 1];
       [2; 1; 2; 2]; [2; 2; 1; 1]; [2; 2; 1; 2];
       [2; 2; 2; 1]; [2; 2; 2; 2]]
```

Exercice 2 [**Tri par sélection d'un tableau**] Sur un tableau, le principe du tri par sélection est le suivant :

- rechercher le plus petit élément du tableau, et l'échanger avec l'élément d'indice 0 ;
- rechercher le second plus petit élément du tableau, et l'échanger avec l'élément d'indice 1 ;
- continuer de cette façon jusqu'à ce que le tableau soit entièrement trié.

Cet algorithme très simple à programmer, mais pas forcément très efficace, est un algorithme « **en place** », c'est à dire que le tableau est directement modifié (il n'y a pas de copie).

1. Écrire une fonction `indice_mini : 'a array -> int -> int` telle que `indice_mini tab deb` renvoie l'indice du plus petit élément du tableau `tab` en ne s'intéressant qu'aux indices à partir de `deb`. Par exemple :

```
# indice_mini [|3;1;2;6;4;5|] 0;;
- : int = 1
# indice_mini [|3;1;2;6;4;5|] 3;;
- : int = 4
```

2. Ecrire une fonction `trie : 'a array -> 'a array` qui reçoit un tableau et envoie ce tableau trié selon l'algorithme présenté.
3. Pour un tableau de n éléments, combien de tests son réalisés lors de l'appel de la fonction `trie`

Exercice 3 [`join` et `split` en OCaml] Il existe en Python deux fonctions très pratiques quand on travaille avec des chaînes de caractères : `join` et `split`. La première permet de concaténer une liste de chaînes en les reliant par un séparateur :

```
>>> "---".join(["abc", "de", "fghi"])
'abc---de---fghi'
```

La deuxième permet à l'inverse de séparer une chaîne en plusieurs parties selon un séparateur :

```
>>> "ab;cdef;ghi".split(";")
['ab', 'cdef', 'ghi']
```

On souhaite reprogrammer ces fonctions en OCaml.

1. Écrire une fonction `joindre` de type `string -> string list -> string` qui, recevant un séparateur (une chaîne de caractères) et une liste de chaînes de caractères, renvoie la chaîne obtenue en concaténant les chaînes de la liste reliées par le séparateur :

```
# joindre "**" ["ab"; "cdef"; "ghi"];;
- : string = "ab**cdef**ghi"
```

2. Écrire une fonction `scinder` de type `string -> char -> string list` qui, recevant une chaîne de caractères et un séparateur (un caractère ici), renvoie la liste obtenue en séparant la chaîne selon le séparateur :

```
# scinder "abc;def;ghij" ';';;
- : string list = ["abc"; "def"; "ghij"]
```

Exercice 4 [**Un dernier cadeau pour ceux qui révisent**] 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 si celle-ci est récursive, en terme de tour de boucle si celle-ci est itérative) ?
3. On cherche à déterminer un algorithme du type "diviser pour régner" pour calculer les termes de cette suite.
   a. Démontrer la relation : $\forall n, p \geqslant 1,\ F_{n+p} = F_n F_p + F_{n-1} F_{p-1}$.
   b. En déduire une expression de F_{2n} et F_{2n+1} en fonction de F_n et F_{n-1}.
   c. Écrire une fonction `fibo_DpR` qui calcule F_n selon la méthode diviser pour régner
4. On souhaite estimer la complexité de cette dernière méthode. On note $T(n)$ le nombre d'appels à la fonction `fibo_DpR` pour calculer F_n.
   a. Calculer $T(0)$, $T(1)$ et $T(2)$.
   b. Si $\alpha_k = T(2^k)$, exprimer α_k en fonction de α_{k-1} pour $k \in \mathbb{N}^\star$.
   c. En déduire $T(n)$ en fonction de n si n est une puissance de 2. Conclure.
   d. (BONUS) : Donner une estimation de $T(n)$ en fonction de n dans le cas général.

Correction 1

1. Une première solution récursive :

```
let rec colle l1 l2 = match (l1, l2) with
          [], _ -> []
        | [h1], [] -> []
        | [h1], (h2::t2) -> (h1::h2) :: (colle [h1] t2)
        | h1::t1, l2 -> (colle [h1] l2) @ (colle t1 l2)
;;
```

Une autre, non récursive mais qui utilise une fonction auxiliaire récursive (pour conserver `l2` en mémoire) :

```
let colle l1 l2 =
        let rec colleR la lb =
                match la, lb with
                        | [], _     -> []
                        | h::t, [] -> colleR t l2
                        | l, h::t  -> (List.hd l::h)::colleR l t
        in colleR l1 l2
;;
```

2. Une solution :

```
let rec mots alphabet = function
        | 0 -> [[]]
        | n -> colle alphabet (mots alphabet (n-1))
;;
```

Correction 3 1. Une solution (regarder ce qu'il se passe si on ne mets pas le second filtrage) :

```
let rec joindre sep = function
        | [] -> ""
        | [h] -> h
        | h::t -> h ^ sep ^ (joindre sep t)
;;
```

2. Une solution : on avance dans le mot caractère par caractère et lorsque l'on rencontre le caractère de séparation, on dépose la chaîne dans la liste et on recommence :

```
let scinder chaine sep =
        let rec avance acc = function
                | i when i = String.length chaine -> [acc]
                | i when chaine.[i] = sep -> acc::avance "" (i+1)
                | i -> avance (acc ^ Char.escaped chaine.[i]) (i+1)
        in avance "" 0
;;
```

▷ Remarque : il existe déjà ces deux fonctions ayant les mêmes effets dans le module `String` : `split_on_char : char -> string -> string list` et `concat : string -> string list -> string`

Correction 4

1.

```
let rec fibo_naif = function
        | 0 | 1 -> 1
        | n -> fibo_naif (n-1) + fibo_naif (n-2)
;;
```

Cet algorithme a une complexité (en nombre d'appels) de $O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$ ce qui n'est pas terrible.

2. On peut le faire de manière itérative :

```
let fibo_suites n =
        let u = ref 1 and v = ref 1 and temp = ref 0 in
                for i = 1 to n do
                        temp := !u;
                        u := !v;
                        v := !temp + !v;
                done;
        !u
;;
```

ou de manière récursive :

```
let fibo_suites n =
        let rec fiboR u v = function
                | 0 -> u
                | n -> fiboR v (u+v) (n-1)
        in fiboR 1 1 n
;;
```

Dans les deux cas, la complexité est clairement en $O(n)$.

3. a. Cette question se fait par récurrence.
   b. Pour $n = p$, on trouve $F_{2n} = F_n^2 + F_{n-1}^2$,
   Pour $p = n + 1$, on trouve $F_{2n+1} = F_n^2 + 2F_nF_{n-1}$.
   c. Attention de bien mettre en mémoire `fibo_DpR (n/2)` et `fibo_DpR (n/2-1)` pour ne pas les recalculer plusieurs fois !

```
let rec fibo_DpR = function
        | 0 | 1 -> 1
        | n when n mod 2 = 0 -> let f = fibo_DpR (n/2) and
                g = fibo_DpR (n/2-1) in f*f + g*g
        | n -> let f = fibo_DpR (n/2) and g = fibo_DpR (n/2 -1)
                in f*f+2*f*g
;;
```

   d. Notons que quelque soit la parité de n le nombre d'appels est identique. Supposons donc que $n = 2^k$ et notons $C(n)$ le nombre d'appels à la fonction lorsque n est entré en paramètre. On a grossièrement $T(n) = 1 + 2T(n/2)$, ainsi $T(n) = O(n)$ (Preuve sur demande).

4. Voici une solution (pour les premières valeurs on a quelque chose de trop rapide pour être mesuré, d'où la boucle TANT QUE :

```
let temps f n =
   let res = Array.make (n+1) 0. and t = ref 0.
      and rien = ref 1 and nb = ref 1 in
      for k = 0 to n do
         t := Sys.time();
         nb := 0;
         while Sys.time() -. !t < 0.0001 do
            rien := f k;
            nb := !nb + 1
         done;
         res.(k) <- (Sys.time() -. !t) /. float_of_int !nb
      done;
      res
;;
```

5.

```
let ecris_tab t1 t2 t3 =
   let oc = open_out "temps.csv" in
      for i = 0 to Array.length t1 - 1 do
         output_string oc (string_of_float (t1.(i)*.100000.));
         output_string oc ";";
         output_string oc (string_of_float (t2.(i)*.100000.));
         output_string oc ";";
         output_string oc (string_of_float (t3.(i)*.100000.));
         output_string oc "\n"
      done;
   close_out oc
;;

let nb = 20 in ecris_tab (temps fibo_naif nb)
                 (temps fibo_suites nb) (temps fibo_DpR nb);;
```