Devoir surveillé n°1

MPSI Lycée Clemenceau : option informatique

Lundi 25 mars 2024

Vous avez 3 heures.

Il sera tenu compte de la présentation et de la rédaction. Les programmes (ou fonctions) devront être commentés et le plus souvent justifiés (terminaison et correction).
Lire l'annexe pour les fonctions prédéfinies et autorisées pour chaque exercice.

Exercice 1 :

Soit $n \in \mathbb{N}^*$, on calcule le produit $prod(n)$ de ses chiffres en base 10, puis le produit des chiffres de $prod(n)$ dans son écriture en base 10 et on recommence l'application de $prod$ jusqu'à obtenir un chiffre entre 0 et 9. Le nombre minimal de fois où on applique $prod$ pour transformer n en un chiffre compris entre 0 et 9 est appelé persistance de n.

1) Donner, en la justifiant, la persistance de 9, puis de 97, puis de 9575.
Correction : après calculs (à condition de savoir ses tables de multiplications) on obtient que la persistance de 9 est 0, (c'est déjà un nombre inférieur ou égal à 9)
celle de 97 est de 3 ($97 \to 63 \to 18 \to 8$),
celle de 9575 est de 5 ($9575 \to 1575 \to 175 \to 35 \to 15 \to 5$).

2) Ecrire une fonction récursive `prod` qui renvoie le produit des chiffres d'un entier écrit en base 10.
Correction : On utilise le fait que le produit des chiffres d'un nombre n est le nombre s'il est inférieur à 10, ou le produit du chiffre des unités (reste de la division euclidienne de n par 10) et le produit des chiffres du quotient de la division euclidienne de n par 10.

```
let rec prod n = match (n/10) with
      | 0 -> n (* le nombre est strictement inférieur à 10*)
      | p -> (n mod 10)*prod(p);; (*on multiplie le chiffre des unités par le
produit du quotient de n par 10*)
```

3) Ecrire une fonction récursive `persi` qui renvoie la persistance d'un entier naturel.
Correction : Il suffit d'appliquer la définition. On peut remarquer que la suite d'entiers obtenue par l'itération de la fonction `prod` est décroissante car pour $n = \sum_{k=0}^{p} a_k 10^k$, avec $a_p \neq 0$,
on a $prod(n) = \prod_{k=0}^{p} a_k < a_p * 10^p < n$. Le programme se termine donc bien.

```
let rec persi n = if n < 10 then 0 (* ne pas faire prod avant d'avoir tester le nombre lui
même*)
      else 1+ persi (prod n);;
```

4) Ecrire une fonction `persi5` qui renvoie le plus petit entier naturel de persistance 5. On proposera une version itérative et une version récursive.
Correction : On parcourt les entiers en commençant à 1. Mais on pourrait commencer à 10 sachant qu'avant la persistance est nulle.
(* version itérative*)

```
let persi5()= let n = ref 1 in
      while persi !n <>5 do incr n done;
      !n;;
```

(* version récursive*)

```
let persi5b () = let rec aux n = match persi n with
               | 5 -> n
               | _ -> aux (n+1)
               in aux 1;;
```

(*dans les deux versions on incrémente un entier pour le tester*)

Exercice 2 : approximation de réels

A) Racine carrée entière

On s'intéresse au problème suivant : étant donné un entier $n \geq 1$, on souhaite déterminer un entier positif s_n de sorte que s_n soit "proche" de $\sqrt{n}$. Nous étudierons deux approches pour résoudre ce problème (*autres que celui vu, mais pas par tous, en td*).

1) Algorithme naïf

Dans cette question on cherche à déterminer le plus grand entier s_n tel que $s_n^2 \leq n$. Pour cela on passe en revue les entiers $1, 2, 3, ..$ jusqu'à trouver s_n.

(a) Ecrire une fonction isqrt_naif prenant n en argument et renvoyant la valeur de s_n en utilisant le principe décrit ci-dessus.

Correction : on utilise le principe demandé

```
let isqrt_naif n = let r = ref 1 in
    while !r * !r <= n do
       incr r
    done;
    !r-1;;
```

Il faut faire attention au fait que pour sortir de la boucle on doit dépasser la valeur cherchée et donc le résultat final est `!r-1` (cela justifie la correction du programme). Le programme se termine car la suite définie par `r` est strictement croissante et majorée par $\lfloor\sqrt{n}\rfloor$.

(b) Déterminer la complexité de votre programme.

Correction : que l'on compte le nombre de comparaisons ou de multiplications (faites pour la comparaison) il y a autant que de passages dans la boucle qui est en $\sqrt{n}$ (nombre de valeurs prises par `r`). La complexité est donc $\Theta(\sqrt{n})$.

2) Une méthode dichotomique

Dans cette question, on reprend le problème de la question précédente. Afin d'obtenir une méthode plus efficace, on considère le programme suivant :

```
let isqrt n = let x =ref 1 and y = ref n in
        while (!y - !x)>1 do
            let z = (!x + !y)/2 in
            if z*z >n then y := z else x := z
        done;
        x;;
```

Si $k \geq 1$ on note x_k et y_k les valeurs des variables x et y à la fin de la $k-$ième itération de la boucle `while`. On convient que $x_0 = 1$ et $y_0 = n$.

(a) Expliciter les exécutions du programme ci-dessus sur l'entier 15. (On donnera en particulier les valeurs successives des variables x, y et z).

Correction : pour $n = 15$, on initialise $x = 1$ et $y = 15$.
on a $y - x > 1$, on rentre dans la boucle : on pose $z = (x + y)/2 = 8$
  on a $z * z > 15$ donc on change $y = 8$ (on a toujours $x = 1$)
on a $y - x > 1$, on rentre dans la boucle : on pose $z = (x + y)/2 = 4$ (attention on calcule le quotient de la division euclidienne, ce sont toujours des entiers).
  on a $z * z > 15$ donc on change $y = 4$ (on a toujours $x = 1$)
on a $y - x > 1$, on rentre dans la boucle : on pose $z = (x + y)/2 = 2$
  on a $z * z \leqslant 15$ donc change $x = 2$ et on garde $y = 4$
on a $y - x > 1$, on rentre dans la boucle : on pose $z = (x + y)/2 = 3$
  on a $z * z \leqslant 15$ donc change $x = 3$ et on garde $y = 4$
on a $y - x = 1$, on ne rentre pas dans la boucle, le programme donne $x = 3$

(b) Démontrer la terminaison du programme.

Correction : dans le cas où $n = 1$ on a $x = y = 1$ et donc le programme ne rentre pas dans la boucle et donne tout de suite le résultat.
Si $n > 1$ on a initialement $x < y$. On pose $z = \lfloor \frac{x+y}{2} \rfloor$, or $x < \frac{x+y}{2} < y$ et donc $x \leqslant z < y$.
Si $x = z$ alors, par définintion de la partie entière $x + 1 > \frac{x+y}{2}$, on en déduit alors $y - x < 1$ le programme c'est alors arrêté avant. Dans le cas contraire $x < z < y$.
On a alors $z - x < y - x$ et $y - z < y - x$. La suite des valeurs prises par $y - x$ est alors une suite strictement décroissante d'entiers. Par propriété de $\mathbb{N}$ elle atteint une valeur dans $\{0, 1\}$ et donc le programme se termine.

(c) Justifier la correction du programme.
Correction : on suppose que $n \neq 1$ (le cas $n = 1$ étant immédiat). On a alors $1 < \sqrt{n} < n$, et donc $1 < n < n^2$, d'où initialement $x^2 < n < y^2$. Par le choix `if z*z >n then y := z else x := z` on aura toujours, $x^2 \leqslant n < y^2$ et donc $x \leqslant \sqrt{n} < y$. A la sortie de la boucle on a $0 \leqslant y - x <= 1$ et donc $x \leqslant \sqrt{n} \leqslant x + 1$. `x` est donc bien la valeur cherchée.

(d) Réécrire `isqrt` en utilisant la récursivité.
Correction : on applique le même principe que le programme itératif

```
let isqrt n =
     let rec aux x y = match y-x with
        | 0 -> x
        | 1 -> x
        | _ -> let z=(x+y)/2 in
               if z*z > n then aux x z else aux z y
     in aux 1 n;;
```

B) Moyenne arithméticogéométrique

3) Soit a et b deux réels tels que $0 < a < b$. Soient (u_n) et (v_n) les suites définies par

$$u_0 = a,\ v_0 = b, \quad \forall n \in \mathbb{N},\ u_{n+1} = \sqrt{u_n v_n},\ v_{n+1} = \frac{v_n + u_n}{2}$$

On admet que ces suites sont adjacentes et on note ℓ leur limite commune appelée moyenne arithméticogéométrique de a et b.
Proposer une fonction `calcul_de_l : float * float * float -> float` qui reçoit en entrée un triplet de réels strictement positifs a, b, p et qui renvoie une valeur approchée de ℓ à p prés.
Correction : il faut faire attention au fait que la fonction demandée n'est pas currifiée mais fait appel à un triplet.
Les suites étant adjacentes avec $u_0 < v_0$ on a toujours $u_n < v_n$.

```
let calcul_de_l (a,b,p)=let u=ref a and v=ref b and aux =ref a in
           while  !v-. !u> p do
               aux:=!u;
               u:=sqrt(!u*. !v);
               v:=(!aux+. !v)/.2.;
           done;
      (!u+. !v)/. 2.;;
```

Prendre la moyenne à la fin n'est pas une obligation car on a un écart maximum de `p` et $u_n < \ell < v_n$, on pourrait donc prendre `!u` pour être sûr d'avoir une valeur par défaut.
Version récursive :

```
let calcul_de_l (a,b,p) = let rec aux u v = if  v-.u <= p then (u +. v)/. 2.
                                else aux (sqrt(u *. v)) ((u +. v)/. 2.)
                                in aux a b;;
```

4) Soit x un réel. On définit trois suites $(s_n)_{n\in\mathbb{N}}$, $(t_n)_{n\in\mathbb{N}}$ et $(c_n)_{n\in\mathbb{N}}$ par :

$$s_0 = \frac{1}{2}\left(x - \frac{1}{x}\right),\ c_0 = \frac{1}{2}\left(x + \frac{1}{x}\right), \quad \forall n \in \mathbb{N}\ c_{n+1} = \sqrt{\frac{1 + c_n}{2}},\ s_{n+1} = \frac{s_n}{c_{n+1}},\ t_n = \frac{s_n}{c_n}$$

On admet que, pour tout $n \in \mathbb{N}$, $t_n \leqslant \ln(x) \leqslant s_n$ et que $(s_n)_{n\in\mathbb{N}}$ et $(t_n)_{n\in\mathbb{N}}$ convergent vers $\ln(x)$.
Proposer un algorithme `calcul_de_ln : float -> float -> float` qui reçoit en entrée deux réels x et $p > 0$ et qui renvoie une valeur approchée de $\ln(x)$ à p près.
Correction : ici la fonction demandée est currifiée, il ne faut donc pas mettre de couple dans l'appel.
Attention aussi à la définition de t_n qui doit se faire après celles de c_n et s_n et non c_{n+1} et s_{n+1}. On peut aussi se passer de poser `t`.

```
let calcul_de_ln x p=let s=ref ((x-. 1./.x)/. 2.) and c=ref ((x+. 1./. x)/. 2. ) in
                     while (!s-. !s/. !c)> p do
                        c:=sqrt((1.+. !c)/. 2.) ;
                        s:=!s/. !c ;
                     done ;
                    (!s+. !s/. !c)/. 2.;;
```

Même remarque pour la fin que pour la question précédente.

Version récursive :

```
let calcul_de_ln x p = let rec aux s c = if (s-s/c) <= p then (s+. s/.c)/.2.
                          else let z = sqrt((1.+. c)/.2.) in aux (s/.z) z
                           in aux  ((x-. 1./.x)/. 2.) ((x+. 1./. x)/. 2. );;
```

Exercice 3

Le but de l'exercice est d'engendrer l'ensemble $\mathscr{P}(E)$ des parties d'un ensemble fini E donné.

Les algorithmes étudiés peuvent s'appliquer à tout type d'ensemble, mais pour leur programmation explicite, on se limitera au cas d'ensembles d'entiers, dont les éléments seront stockés dans un tableau `ens`.

Pour caractériser un sous-ensemble A d'un tel ensemble E, on convient d'utiliser un *tableau de présence* noté `p` de la façon suivante : pour tout `k` entre 1 et n, `ens.(k)` est élément de A si et seulement si `p.(k)=1`.

Exemple, si $n = 4$ et si le tableau `ens` est `[|3;5;7;9;|]`, alors $E = \{3, 5, 7, 9\}$ et :
- `p=[|0;1;0;0|]` correspond à la partie $\{5\}$
- `p=[|1;0;0;1|]` correspond à la partie $\{3, 9\}$, etc.

Pour les fonctions et procédures demandées, le candidat pourra choisir entre version itérative ou récursive lorsque l'énoncé ne précise rien.

Si une version récursive est requise, la fonction ou procédure écrite pourra être elle-même récursive, ou bien faire appel à une fonction ou procédure auxiliaire récursive.

1) *Affichage d'une partie* : écrire deux versions (itérative **et** récursive) d'une procédure

`affpartie ens p`

qui affiche à l'écran la partie de l'ensemble E (de cardinal n, stocké dans le tableau `ens`) caractérisée par le tableau de présence `p`. Les valeurs seront affichées entre accolades, séparées par des espaces.

Par exemple, des paramètres correspondant à la partie $\{3, 9\}$ devront conduire à l'affichage de

```
{ 3 9 }
```

Correction : Pour une version itérative, pas de difficulté particulière. On balaye l'ensemble des éléments de E, à savoir les `ens.(k)` pour `k` allant de `0` à `n-1` , et l'on écrit la valeur si et seulement si `p.(k)=1`. Penser à écrire un espace après chaque valeur de la partie à afficher. Justification banale.

```
let affpartie ens p =
    let n = Array.length ens in
      print_string "{ "
      for i=0 to n-1 do
        if p.(i)=1 then begin print_int ens.(i); print_string " " end done;
      print_string "}"
```

Pour une version récursive, attention de ne pas écrire une paire d'accolades à chaque appel récursif ! Une fonction auxiliaire récursive se charge d'écrire les éléments suivis d'un espace, le corps de la fonction principale écrit une seule paire d'accolades :

```
let affpartier ens p =
    let n = Array.length ens in
      let rec aux k =
        if k > 0 then begin
          aux (k-1);
          if p.(k)=1 then begin print_int ens.(k); print_string " " end
        end;
        in
      print_string "{ ";
      aux (n-1);
      print_string "}"
```

2) *Génération de $\mathcal{P}(E)$ – version itérative* : pour cette première méthode, on utilise la propriété suivante, **que l'on ne demande pas de démontrer**. Lorsqu'un entier j varie de 0 à $2^n - 1$, son écriture en base 2 fournit tous les tableaux de présence correspondant aux 2^n parties d'un ensemble à n éléments (sous réserve bien sûr de considérer les écritures comportant exactement n chiffres, avec éventuellement des 0 pour les chiffres de poids le plus élevé). Par exemple, pour $n = 3$, les écritures en base 2 des entiers de 0 à 7 sont : 000, 001, 010, 011, 100, 101, 110, 111.

a) Écrire une fonction

`deuxpuis n`

renvoyant 2^n.

Correction : j'ai choisi la version récursive de l'exponentiation rapide mais on pouvait faire une simple boucle ou une fonction simple récursive.

```
let rec deuxpuiss n = match n with
        | 0 -> 1
        | n when n mod 2 = 0 -> let m = deuxpuiss (n/2) in m*m
        | _ -> let m = deuxpuiss (n/2) in 2*m*m;;
```

b) On convient ici que les valeurs (0 ou 1) stockées dans un tableau `p` de présence correspondent aux chiffres de l'écriture en base 2 d'un entier j, **rangés par ordre de poids croissant avec l'indice**.

Par exemple, `p=[|1;1;1;0;0;1;0|]` correspond à $j = 39 = 2^0 + 2^1 + 2^2 + 2^5$ et `p=[|0;0;0;1;0;1|]` correspond à $j = 40 = 2^3 + 2^5$.

Écrire deux versions (itérative **et** récursive) d'une procédure d'en-tête

`ajoute1 p`

recevant un tableau `p` contenant — selon le principe ci-dessus — les chiffres de l'écriture en base 2 d'un entier j et modifiant `p` pour qu'il représente de même l'entier $j + 1$.

On justifiera la méthode utilisée pour effectuer l'addition.

Correction : L'exemple de l'énoncé était censé donner l'idée : ajouter 1 à un entier écrit en base 2 se fait en parcourant les chiffres de ladite écriture, à partir de celui de poids le plus faible, à la recherche du premier 0, que l'on remplace par un 1, tous les 1 rencontrés précédemment étant remplacés par des 0. Ce principe découle de l'identité

$$\sum_{j=0}^{k-1} 2^j = \frac{1-2^k}{1-2} = 2^k - 1$$

qui se traduit par l'écriture en base 2

$$\underbrace{\overline{11\ldots1}}^{2} + 1 \quad = \quad \overline{1\underbrace{00\ldots0}}^{2}$$
$$k \text{ fois } 1 \qquad\qquad k \text{ fois } 0$$

Version itérative :

On applique ici le principe décrit ci-dessus, en remplaçant au fur et à mesure par des 0 les 1 des chiffres de poids croissants, jusqu'à trouver le premier 0 qu'on remplace par un 1. La boucle se termine bien, puisque si `p` ne contenait que des 1, c'est qu'il représenterait l'entier $j = 2^{NMax} - 1$, ce qui est exclu.

```
let ajout1 p =
        let k = ref 0 in
          while p.(!k)=1 do
            p.(!k)<- 0;
            incr(k);
            done;
          p.(!k)<-1;;
```

Version récursive : Même principe pour une version récursive. Les appels récursifs se terminent au premier 0 rencontré, ce qui arrive nécessairement comme on vient de le voir.

```
let ajout1R p =
        let rec aux k =
          if p.(k)=1 then begin
            p.(k)<-0;
            aux (k+1);
            end
          else p.(k)<- 1
        in
        aux 0;;
```

c) Déduire des questions précédentes une procédure itérative

`AffEnsPartiesI E`

qui affiche à l'écran successivement toutes les parties de l'ensemble E (stocké dans le tableau `E`).

Correction : il suffit d'appliquer le principe décrit dans l'énoncé, qui justifie ce programme :

```
let affenspartiesI e =
        let n = Array.length e in
        let p = Array.make n 0 in
          print_string " "; (* ensemble vide *)
          for j=1 to deuxpuiss n - 1 do
            ajout1 p;
            affpartie e p;
            done;;
```

3) *Génération de $\mathcal{P}(E)$ – version récursive* : on peut engendrer récursivement tous les tableaux de présence correspondant aux différentes parties de l'ensemble donné E (et afficher la partie A associée au tableau `p` dès qu'il est rempli !). Écrire selon ce principe une fonction récursive ou utilisant une fonction auxiliaire récursive

```
affenspartiesr ens
```

qui affiche à l'écran successivement toutes les parties de l'ensemble E .

Cette question est indépendante des précédentes.

Correction :

```
let affenspartiesr ens =
      let n = Array.length ens in
      let p = Array.make n 0 in
      let rec aux k =
        if k = -1 then affpartier ens p
          else begin
            p.(k)<- 0;aux (k-1);
            p.(k)<- 1;aux (k-1);
            end;
        in
      aux (n-1);;
```

La fonction récursive `aux` remplit récursivement le tableau `p` comme le suggérait l'énoncé.

On part de `k=n-1` et `k` décroît strictement jusqu'à la valeur `-1` au fil des appels récursifs.

Lorsque `k=-1`, d'une part les appels récursifs se terminent (!), d'autre part on affiche la partie correspondant au tableau de présence `p` (l'astuce consiste à afficher à chaque fois la partie "complète" avec bien sûr la valeur `p` pour second paramètre de `affpartier`, en ayant pris soin d'avoir un tableau `p` "global" pour tous les appels récursifs).

Plus formellement :

- La terminaison du programme est justifiée par la décroissance stricte de `k` au fur et à mesure des appels récursifs et leur arrêt pour `k=-1`.
- Sa correction se prouve par exemple en établissant par récurrence sur $k \in [\![-1, n-1]\!]$ la propriété $\mathcal{P}_k$ suivante :
  "l'appel de `aux k` entraîne l'affichage de l'ensemble des parties de $E = \{e_0, \ldots, e_{n-1}\}$ dont l'intersection avec $\{e_{k+1}, \ldots, e_{n-1}\}$ est décrite par les valeurs de `p.(k+1)`,..., `p.(n-1)` au moment de l'appel" (où l'on a noté, pour tout j, e_j l'élément de E stocké dans `e.(j)`).

Exercice 4 : manipulation de listes

1) Ecrire une fonction calculant la somme des éléments (entiers) d'une liste.

Correction : Première version non terminale

```
let rec somme1 lst = match lst with
             | [] -> failwith "liste vide"
             | [a] -> a
             | t::q -> t + somme1 q;;
```

Version récursive terminale

```
let somme lst = match lst with
            | [] -> failwith "liste vide"
            | t::q -> let rec aux l s = match l with
```

```
                    | [] -> s
                    | t::q -> aux q (s+t)
            in aux q t;;
```

2) Ecrire une fonction `sublist : 'a list -> int -> int -> 'a list` renvoyant la liste formée des éléments dont la position est comprise, au sens large, entre les valeurs spécifiées.

Correction : il faut gérer le fait que la liste soit assez grande sans utiliser `List.length`. On commence par éliminer les éléments en tête de la liste puis ceux de la fin.

```
let sublist lst d f =
        let rec aux_debut i ls = match (d-i,ls) with
            | (k,[]) -> failwith "liste trop courte"
            | (0,ls) -> ls
            | (k,a::q) -> aux_debut (i+1) q
        in
        let rec aux_fin j ls = match (f-d-j,ls) with
            | (k,[]) -> failwith "liste trop courte"
            | (0,a::q) -> [a]
            | (k,a::q) -> a::(aux_fin (j+1) q)
        in
        let ls = aux_debut 0 lst in
        aux_fin 0 ls;;
```

3) Ecrire une fonction `minmaxlist : 'a list -> 'a * 'a` qui renvoie le couple formé par le plus petit et plus grand élément d'une liste.

Correction : on parcourt la liste

```
let minmaxlist2 lst = match lst with
        | [] -> failwith "liste vide"
        | a::q -> let rec aux lst pp pg = match lst with
                        | [] -> pp,pg
                        | b::q -> aux q (min pp b) (max pg b)
                  in aux q a a;;
```

4) Ecrire une fonction `min2list : 'a list -> 'a * 'a` qui retourne le couple des deux plus petits éléments.

Correction : on part dans l'idée que ces deux éléments sont distincts

```
let rec min2list lst = match lst with
        | [] - > failwith "liste trop courte"
        | [_] -> failwith "liste trop courte"
        | [x;y] -> (min x y, max x y)
        | x::q -> let (a,b) = min2list q in
            if b < x then (a,b) else if a < x then (a,x) else (x,a);;
```

5) Ecrire une fonction `listplateau : 'a list -> int * int * 'a` identifiant le plus long plateau formé d'éléments consécutifs identiques dans une liste quelconque (donc pas forcément triée !). Le résultat est le triple `(d; l; x)` formé par la position `d` du début du plateau, par sa longueur `l` et par la valeur `x` de l'élément ainsi répété.

Par exemple :

```
listplateau ["e"; "x"; "x"; "x"; "c"; "c"; "x"; "c"; "c"; "c"; "c"; "u"; "c"];;
- : int * int * string = 7, 4, "c"
```

Correction : On commence par le cas d'une liste d'un seul élément. Puis on va chercher le plus grand plateau de la queue de la liste.

On regarde si l'élément de la tête de la liste complète un plateau `if x = (List.hd q)`, dans ce cas on compare sa longueur avec le plus grand plateau. Si c'est la même `lc = lp` il est susceptible d'augmenter donc on le note comme le plus grand `(lc + 1, 0, lc + 1, x)`, sinon le plus grand a une position décalée d'un `(lc + 1, dp + 1, lp, a)`.

Si `x <> (List.hd q)` il commence un plateau de longueur 1 et le plus grand a une position décalée d'un.

```
let listplateau l =
        let rec aux lst = match lst with
            | [] -> failwith "liste vide"
            | [x] -> (1, 0, 1, x)
            | x::q -> let (lc, dp, lp, a) = aux q in
                    if x = (List.hd q) then
                        if lc = lp then (lc + 1, 0, lc + 1, x)
                        else (lc + 1, dp + 1, lp, a)
                    else (1, dp + 1, lp, a)
                in let (lc, dp, lp, a) = aux l
            in (dp, lp, a);;
```

6) Ecrire une fonction `ocsmin : 'a list -> 'a * int = <fun>` qui renvoie le couple formé par l'élément minimum d'une liste et le nombre de fois où il apparaît. Cette fonction ne devra parcourir qu'une seule fois la liste.

Correction :

la fonction qui suit ne parcourt qu'une seule fois la liste.

Elle utilise une fonction auxiliaire dont les paramètres sont une liste un élément qui est le minimum provisoire et son occurrence.

```
let ocsmin lst = match lst with
        | [] -> failwith "liste vide"
        | a::q -> let rec aux lst e n = match lst with
                        | [] -> (e,n)
                        | a::q ->
                        | a::q when a < e -> aux q a 1
                        | a::q when a = e -> aux q a (n+1)
                        | a::q -> aux q e n
                in aux q a 1;;
```

Exercice 5 : Représentation d'ensembles avec des intervalles (CCP 2015)

De nombreux algorithmes reposent sur la manipulation d'ensembles d'éléments ordonnés. Lorsque ces ensembles contiennent de nombreux éléments adjacents (aucune valeur n'existe entre les deux éléments), il est plus performant en terme de temps de calcul et d'occupation mémoire de manipuler des intervalles au lieu de valeurs singulières. Cela permet également de manipuler des ensembles infinis sous la forme d'un ensemble fini d'intervalles contenant un nombre de valeurs infinies (i.e. dans $\mathbb{Q}$ ou $\mathbb{R}$).

L'objectif de ce problème est de comparer deux implantations différentes d'un ensemble d'entiers, la première à base de listes triées d'intervalles, et la seconde à base d'arbres binaires d'intervalles.

Nous nous limiterons à des intervalles fermés de $\mathbb{R}$ dont les bornes sont des entiers $[min, max]$. L'extension à des intervalles ouverts et aux valeurs $-\infty$ et $+\infty$ ne pose pas de problème majeur.

I Préliminaires : Représentation des intervalles

I.1 Définitions

Définition 1) (Intervalles disjoints) Deux intervalles sont disjoints si leur intersection est vide.

Définition 2) (Fusion d'intervalles) La fusion de deux intervalles a comme minimum le plus petit des minima des deux intervalles, et comme maximum le plus grand des maxima des deux intervalles. Cette opération correspond à l'union des intervalles si ceux-ci ne sont pas disjoints.

I.2 Représentation en CaML

Nous ferons l'hypothèse que les couples qui représentent des intervalles sont bien formés, c'est-à-dire que la valeur représentant le minimum est inférieure ou égale a la valeur représentant le maximum.

Un intervalle est représenté par le type `intervalle` équivalent à une paire de `int` (son minimum en premier et son maximum en second).

```
type intervalle = int * int;;
```

Nous allons maintenant définir plusieurs opérations sur les intervalles.

2.1.3 Opérations sur la structure d'intervalle

La première opération est le test si deux intervalles sont disjoints.

Question 1) Écrire en CaML une fonction `disjoints` de type `intervalle -> intervalle -> bool` telle que l'appel (`disjoints i1 i2`) renvoie la valeur `true` si les intervalles `i1` et `i2` sont disjoints, et la valeur `false` sinon.

Correction : Pour savoir si les intervalles $[g_1, d_1]$ et $[g_2, d_2]$ sont disjoints, on vérifie que $d_1 < g_2$ ou que $d_2 < g_1$.

```
let disjoints   ((g1,d1) : intervalle) ((g2,d2) : intervalle) =
                      d2 < g1 || d1 < g2;;
```

La seconde opération est la fusion de deux intervalles.

Question 2) Écrire en CaML une fonction `fusion` de type `intervalle -> intervalle -> intervalle` telle que l'appel (`fusion i1 i2`) renvoie un intervalle correspondant à la fusion de `i1` et `i2`.

Correction : Par définition, la fusion des intervalles $[g_1, d_1]$ et $[g_2, d_2]$ est égale à l'intervalle $[min(g_1, g_2), max(d_1, d_2)]$.

```
let fusion ((g1,d1) : intervalle) ((g2,d2) : intervalle) =
                      ((min g1 g2,max d1 d2) : intervalle);;
```

Remarque : le texte est confus car on ne sait pas si on doit tester avant le fait qu'ils soient disjoints, ce qui empêcherait la fusion.

Voici une version tenant compte de la remarque :

```
let fusion ((g1,d1) : intervalle) ((g2,d2) : intervalle) =
          if disjoints (g1,d1) (g2,d2) then failwith "intervalles disjoints"
            else ((min g1 g2,max d1 d2) : intervalle);;
```

2.2 Réalisation à base d'une liste triée d'intervalles

La réalisation la plus simple d'un ensemble de valeurs en utilisant des intervalles consiste à utiliser une liste d'intervalles. Pour réduire la complexité amortie de certaines opérations, nous utiliserons une liste triée d'intervalles d'entiers.

2.2.1 Définitions

Les algorithmes manipuleront des listes d'intervalles respectant certaines contraintes que nous appellerons liste bien formée.

Définition 3) (Liste bien formée d'intervalles) Une liste bien formée d'intervalles est une liste d'intervalles qui respecte les contraintes suivantes :

- les intervalles sont bien formés,
- les intervalles sont disjoints deux à deux,
- la liste est triée selon la relation d'ordre suivante : un intervalle i_1 est strictement plus petit qu'un intervalle i_2 si et seulement si le maximum de i_1 est strictement plus petit que le minimum de i_2.

On peut montrer qu'une liste triée d'intervalles bien formés, est une liste bien formée d'intervalles.

La définition 3) peut s'exprimer sous la forme d'une propriété $LBFI(e)$ qui indique que $e = \langle v_1, ..., v_n \rangle$ est une liste bien formée d'intervalles (contenant les intervalles v_i) :

$$LBFI(e) \iff \begin{cases} e = \langle\rangle \\ \text{ou } e = \langle u_1 \rangle \\ \text{ou } e = \langle v_1, v_2, ..., v_n \rangle \text{ et } v_1 < v_2 \text{ et } LBFI(e') \text{ avec } e' = \langle v_2, ..., v_n \rangle \end{cases}$$

2.2.2 Représentation en CaML

Une liste triée d'intervalles est représentée par le type `liste` équivalent à une liste d'`intervalle`.

```
type liste = intervalle list;;
```

Nous allons maintenant définir plusieurs opérations sur les listes bien formées d'intervalles et estimer leur complexité.

2.2.3 Opérations sur la structure de liste bien formée d'intervalles

La première opération est l'ajout d'un intervalle dans une liste bien formée d'intervalles.

Question 3) Écrire en CaML une fonction `ajouter` de type `intervalle -> liste -> liste` telle que l'appel (`ajouter i l`) sur une liste bien formée d'intervalles `l` renvoie une liste bien formée d'intervalles contenant les intervalles contenus dans la liste `l` qui sont disjoints de `i`, et :

- soit le résultat de la fusion de `i` et des intervalles contenus dans la liste `l` qui ne sont pas disjoints de `i`,
- soit l'intervalle `i`, si tous les intervalles contenus dans la liste `l` lui sont disjoints.

L'algorithme utilisé ne devra parcourir qu'une seule fois la liste. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.

Correction : Pour ajouter un intervalle i à une liste bien formée d'intervalles, on envisage les différents cas de figure :

- Si la liste est vide, on retourne la liste constituée du seul intervalle i (ligne 2),
- Si l'intervalle i et j le premier intervalle de la liste sont disjoints et que $i < j$, on ajoute i en tête de la liste (ligne 3),
- Si l'intervalle i et le premier intervalle de la liste ne sont pas disjoints, on les fusionnent et on appelle de nouveau la fonction `ajoute` avec le nouvel intervalle et la queue de la liste comme paramètres (ligne 4),
- Le dernier cas correspond au cas où les intervalles i et j sont disjoints et $j < i$ pour l'ordre lexicographique sur les couples d'entiers. On ajoute i dans la queue de la liste (ligne 5),

```
let rec ajouter i lst = match (i,lst) with
          | i , [] -> [i]
          | i , (j::js) when  (snd i) < (fst j) -> i::j::js
          | i , (j::js) when not(disjoints i j) -> ajouter (fusion i j) js
          | i , (j::js) -> j::(ajouter i js);;
```

Question 4) Donner des exemples de valeurs des paramètres `i` et `l` de la fonction `ajouter` qui correspondent aux meilleur et pire cas en nombre d'appels récursifs effectués.

Calculer une estimation de la complexité dans les meilleur et pire cas de la fonction `ajouter` en fonction de la longueur de la liste `l`. Cette estimation ne prendra en compte que le nombre d'appels récursifs effectués.

Correction : Dans le meilleur des cas, un seul appel suffit. C'est le cas où l'élément à ajouter se place en tête de liste. La complexité est alors en $O(1)$.

Exemple : `ajouter (0,1) [(2,3);(4,5);...;(2n,2n+1)]`.

Dans le pire des cas, l'intervalle à ajouter englobe tous les intervalles. À chaque étape, il faut fusionner l'intervalle à ajouter avec le premier intervalle de la liste et recommencer. La complexité est alors en $O(n)$ où n est le nombre d'éléments de la liste.

Exemple : `ajouter (0,2n) [(0,2);(2,3);(4,5);...;(2n,2n+1)]`.

La deuxième opération est le test d'appartenance d'une valeur dans une liste bien formée d'intervalles.

Question 5) Écrire en CaML une fonction `appartenir` de type `int -> liste -> bool` telle que l'appel (`appartenir v l`) sur une liste bien formée d'intervalles `l` renvoie la valeur `true` si la valeur `v` appartient à un des intervalles contenus dans la liste `l` et la valeur `false` sinon.

L'algorithme utilisé ne devra parcourir qu'une seule fois la liste. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.

Correction : On envisage les différents cas de figure.

- Si la liste est vide, c'est que l'élément n'appartient à aucun intervalle (ligne 2), on retourne `false`,
- Si v est strictement plus petit que la borne inférieure du premier intervalle de la liste, c'est que l'élément n'appartient à aucun intervalle (ligne 3), on retourne `false`,
- La ligne 4 correspond au cas $v \in [m, M]$ où $[m, M]$ est le premier intervalle de la liste. L'élément appartient donc à l'un des intervalles, on retourne `true`,

– La ligne 5 correspond au cas $v > M$ où $[m, M]$ est le premier intervalle de la liste. L'élément n'appartient pas au premier intervalle mais peut appartenir à l'un des autres intervalles de la liste. On teste cette possibilité par un appel récursif sur la queue de la liste des intervalles.

```
let rec appartenir v lst = match v,lst with
         | v,[] -> false
         | v,((g,d)::suite) when v < g -> false
         | v,((g,d)::suite) when v <= d -> true
         | v,((g,d)::suite) (* when v > d *) -> appartenir v suite;;
```

La troisième opération est la vérification qu'une liste d'intervalles est bien formée, c'est-à-dire respecte les contraintes de la définition 3) formalisée par la propriété $LBFI(\texttt{e})$ définie précédemment.

Question 6) Écrire en CaML une fonction `verifier` de type `liste -> bool` telle que l'appel (`verifier l`) sur une liste d'intervalles `l` renvoie la valeur `true` si la liste `l` respecte les contraintes d'une liste bien formée d'intervalles, et la valeur `false` sinon. L'algorithme utilisé ne devra parcourir qu'une seule fois la liste. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.

Correction : Pour vérifier qu'une liste est bien formée, on étudie les différents cas de figure possibles :
– La liste est vide, on retourne `true` (ligne 2),
– La liste est composée d'un seul élément, on retourne `true` (ligne 3),
– S'il y a plus de deux éléments (ligne 4), on vérifie
  – que i_1 représente bien un intervalle,
  – que les deux premiers intervalles i_1 et i_2 vérifient, i_1 inférieur strict à i_2
  – la liste sans le premier intervalle est une liste bien formée d'intervalles.

```
let rec verifier (l : liste) = match l with
      | ([] : liste) -> true
      | ([(mi,ma)] : liste) -> mi <= ma
      | (mi1,ma1)::(mi2,ma2)::suite -> (mi1 <= ma1) && (ma1 < mi2)
                  && (verifier ((mi2,ma2)::suite));;
```

Annexe

Exercice 1 :

Fonctions utiles : / pour le quotient de la division euclidienne de deux entiers, `mod` pour le reste de la division euclidienne

Exercice 2 :

Rien de particulier

Exercice 3 :

Fonctions utiles :
- / pour le quotient de la division euclidienne de deux entiers, `mod` pour le reste de la division euclidienne
- `print_string` : affiche une chaine de caractère
- `print_sint` : affiche un entier
- `incr` : fonction qui incrémente d'un une référence entière

Pour les tableaux :
- `Array.make n a` : créer un tableau de taille n dont tous les éléments sont égaux à `a`
- `Array.length` : donne la longueur d'un tableau
- `p.(i)` : donne l'élément du tableau d'indice `i` (l'indice commence à 0)
- `p.(i)<- a` : pour mettre la valeur de `a` dans le tableau à l'indice `i`

Exercice 4 :

Aucune fonction sur les listes prédéfinie n'est autorisée à par, si besoin, `List.hd`, `List.tl`, le constructeur `::`.
Les fonctions de deux variables `min` et `max` sont autorisées. Elles sont curryfiées.

Exercice 5 :

Pour avoir le premier et le second élément d'un couple on utilise `fst` et `snd`.
Aucune fonction sur les listes prédéfinie n'est autorisée à par, si besoin, `List.hd`, `List.tl`, le constructeur `::`.
Les fonctions de deux variables `min` et `max` sont autorisées. Elles sont curryfiées.