Correction de devoir du Vendredi 19 Mars

Exercice 1 (La syntaxe en OCaml)

1. Le type `array` est un type complexe ayant les caractéristiques suivantes :
   — les objets le composant sont tous de même type,
   — chaque objet est directement accessible (accès en temps constant),
   — l'ensemble est de taille fixe (type statique),
   — la valeur de chaque objet peut être modifiée (type mutable).
   Le type `list` est un type complexe ayant les caractéristiques suivantes :
   — les objets le composant sont tous de même type,
   — chaque objet contient l'adresse du suivant et ainsi le temps d'accès à chaque objet n'est plus identique (accès en temps linéaire),
   — l'ensemble est de taille modifiable (type dynamique),
   — la valeur de chaque objet ne peut être modifiée (type non mutable).
2. (a) `- : bool = true`
   (b) `- : int = 0`
   (c) `- : 'a list list = [[]]`
   (d) Erreur car 2 est de type `int` et devrait être de type `float`.
   (e) `- : (int * int) list = [(1, 2); (3, 0)]`
   (f) `- : char = 'f'`
   (g) `- : int array list = [[|2;3|];[|1;2;4|]]`
   (h) `- : 'a array -> int -> int -> 'a array = <fun>`
   (i) `- : int array = [|2; 4|]`
   (j) Erreur car `[1,2;4,5]` n'est pas un tableau.
3. (a)
```
(1. +. sqrt(5.)) /. 2.;;
- : float = 1.6180339887498949
```
   (b)
```
[|1;4;5;2;8|].(2) <- 0;;
- : unit = ()
```
   (c)
```
List.hd (List.tl (List.tl [1;4;5;2;8]));;
- : int = 5
```
   (d)
```
Array.make 3 2;;
- : int array [|2; 2; 2|]
```
4. (a) `x` est de type `int list list` et `List.length ([2;4]::x) ;;` est de type `int`.
   (b) `x` est de type `int list list` et `x@[[11]] ;;` est aussi de type `int list list`.
   (c) `x` est de type `'a` et `x::[] ;;` est de type `'a list` (il y a un polymorphisme de type ici).

Exercice 2 (Du cours...)

1. **Factorielle :**
   (a)

```
let fact_iter n =
  let f = ref 1 in
    for k = 1 to n do
      f := !f * k
    done;
  !f
;;
```

(b)

```
let rec fact_rec n =
  if (n = 0)
    then
      1
    else
      n * (fact_rec (n-1))
;;
```

(c)

```
let rec facto_aux n acc =
   if (n = 0)
      then
         acc
      else
         facto_aux (n-1) (acc*n)
;;

let facto_terminale n=
    facto_aux n 1
;;
```

2. **Suite récurrente :**

(a)

```
let suite_iter n =
  let u = ref 2. in
    for k = 1 to n do
      u := log (1. +. !u)
    done;
    !u
;;
```

(b)

```
let rec suite_rec n =
  if (n = 0)
    then
      2.
    else
      log (1. +. (suite_rec (n-1))
;;
```

(c)

```
let rec suite_aux n acc=
  if (n = 0)
    then
      acc
    else
      suite_aux (n-1) (log (1. +. acc))
;;

let suite_terminale n = suite_aux n 2.0
;;
```

Exercice 3 (Manipulation récursive de tableau en place)

1.

```
let maximum a =
  let p = ref 0 in
    for i = 1 to (Array.length a - 1) do
      if a.(i) > a.(!p) then
          p := i
    done;
  !p
;;
```

2. (a) • Si $i = j$, alors, il n'y a qu'une seule valeur dans le tableau donc $p = i$.

• Si $i < j$, soit un indice q de la valeur maximale parmi les valeurs $a[k]$, k appartenant à $[\![i+1, j]\!]$, c'est-à-dire tel que :

$$a[q] = \max\{a[i+1], a[i+2], \ldots, a[j-1], a[j]\}\,.$$

On compare alors $a[i]$ avec $a[q]$: si $a[i] > a[q]$ alors l'indice p tel que

$$a[p] = \max\{a[i], a[i+1], \ldots, a[j-1], a[j]\}\,.$$

est $p = i$, sinon, le maximum est atteint en $p = q$.

•

```
let rec maximum_aux a i j=
   if (i=j)
     then
       i
     else
       let q = (maximum_aux a (i+1) j) in
          if a.(i) > a.(q)
             then
                i
             else
                q
;;
```

(b)

```
let  maximum a =
   let n=Array.length a in maximum_aux a 0 (n-1)
;;
```

Exercice 4

1.
```
let rec appartenance x e = match e with
  | [] -> false
  | t::q -> (t=x) || (appartenance x q)
;;
```
2.
```
let rec test_ensemble l = match l with
  | [x] -> true
  | t::q -> (not (appartenance t q)) && (test_ensemble q)
;;
```
3.
```
let rec union e1 e2 = match e2 with
  | [] -> e1
  | t::q -> if (appartenance t e1) then union e1 q
                                   else t::(union e1 q)
;;
```
4.
```
let rec intersection e1 e2 = match e2 with
  | [] -> []
  | t::q -> if (appartenance t e1) then t::(intersection e1 q)
                                   else (intersection e1 q)
;;
```
5.
```
let rec difference e1 e2 = match e1 with
  | [] -> []
  | t::q -> if (appartenance t e2) then difference q e2
                                   else t::(difference q e2)
;;
```

Exercice 5

1.
```
let compteur1 n x =
  let r = ref n and a = ref 0 in
    while (!r >= x) do
      r := !r-x;
      a := !a+1
    done;
    !a
;;
```
Cet algorithme permet de faire la division euclidienne de n par x.

2.
```
let compteur2 n x p =
  let r = ref n and a = ref 0 in
    while (!r >= x) && (!a < p) do
      r := !r-x;
      a := !a+1
    done;
    !a
;;
```

3.
```
let compteur3 n x p =
  let r = ref n and a = ref 0 in
    while (!r >= x) && (!a < p) do
      r := !r-x;
      a := !a+1
    done;
    if (!r < x) then !a else -1
;;
```

4. (a) — Le problème a toujours une solution. En effet, si une somme de n cts nous est demandée, une solution triviale est de donner n pièces de 1 cts.
— Le problème a également toujours une solution en suivant l'algorithme glouton. Démontrons le par récurrence. Notons $(\mathcal{H}_n)$ la propriété "l'algorithme glouton admet une solution pour une somme de n cts".
• Il est clair que $(\mathcal{H}_n)$ est vraie pour $n = 0, 1, 2$.
• Soit $n \in \mathbb{N}^*$. Supposons que $(\mathcal{H}_0), \dots, (\mathcal{H}_{n-1})$ sont vraies et montrons que $(\mathcal{H}_n)$ est vraie.
Soit x la plus grande valeur de billets ou pièces telle que $x \leqslant n$ et posons $a = \left\lfloor \frac{n}{x} \right\rfloor$ (partie entière de la fraction $\frac{n}{x}$). Alors $a \leqslant \frac{n}{x} < a + 1$ donc $ax \leqslant n < ax + x$ et $0 \leqslant n - ax < x \leqslant n$.
Comme $n - ax < n$, on sait par hypothèse de récurrence qu'il existe une solution du problème en suivant l'algorithme glouton pour la somme $n - ax$ (sans billet ou pièce d'une valeur x car $n - ax < x$). On en déduit une solution du problème en suivant l'algorithme glouton pour la somme n en ajoutant a billets ou pièces d'une valeur x à la monnaie à rendre obtenue par la récurrence. Donc $(\mathcal{H}_n)$ est vraie.
• Par le principe de récurrence, $(\mathcal{H}_n)$ est vraie pour tout $n \in \mathbb{N}$.

(b)

```
let x=[|5000;2000;1000;500;200;100;50;20;10;5;2;1|];;

let gloutonillimite n x =
  let c = Array.make 12 0 and s = ref n and k = ref 0 in
    while (!s <> 0) do
      let aux = compteur1 !s x.(!k) in
        c.(!k) <- aux;
        s := !s - aux * x.(!k);
        k := !k+1
    done;
    c
;;
```

5. (a)

```
let sommemax x p =
  let s = ref 0 in
    for k=0 to 11 do
      s := !s+p.(k)*x.(k)
    done;
    !s
;;
```

Une somme inférieure à cette somme maximale ne peut pas toujours être constituée avec nos pièces et billets disponibles. Par exemple, si nous disposons d'un seul billet de 50 euros, on ne pourra pas constituer une somme de 20 euros.

(b)

```
let gloutonechec x n p =
  let c = Array.make 12 0 and s = ref n and k = ref 0
         and a = ref 0 in
    while (!s <> 0) && (!a <> -1) do
      a := compteur3 !s x.(!k) p.(!k) ;
      if (!a <> -1) then
        begin
          c.(!k) <- !a;
          s := !s - !a * x.(!k);
          k := !k+1
        end
    done;
    if (!a <> -1) then c
                  else failwith "Probleme de monnaie !"
;;
```

(c)

```
let gloutonreel x n p =
  let c = Array.make 12 0 and s = ref n and k = ref 0
          and a = ref 0 in
    while (!s <> 0) && (!k <= 11) do
      a := compteur2 !s x.(!k) p.(!k) ;
      c.(!k) <- !a;
      s := !s - !a * x.(!k);
      k := !k+1
    done;
    if (!s=0) then c else failwith "Probleme de monnaie !"
;;
```

6.

```
let controle x n c p=
  let test = ref true and s = ref 0 in
    for k=0 to 11 do
      test := (!test) && (c.(k) <= p.(k));
      s := !s + c.(k) * x.(k)
    done;
    !test && (!s=n)
;;
```

7. Avec l'ancien système monétaire du Royaume-Uni, l'algorithme glouton n'est pas toujours le plus efficace en termes de nombre de billets et pièces manipulés pour constituer une somme donnée.

   Par exemple, la somme 48 se décompose avec l'algorithme glouton sous la forme $48 = 1 \times 30 + 1 \times 12 + 1 \times 6$ donc avec 3 pièces. Mais $48 = 2 \times 24$ et on peut donc utiliser seulement 2 pièces.