Correction du l'Interrogation 0 du Lundi 1 Septembre

Exercice 1 (La syntaxe OCaml)

(a) `- : bool = true`
(b) `- : int = 2`
(c) `- : 'a 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 = 'm'`
(g) `- : int list = []`
(h) `- : 'a list -> int = <fun>`
(i) `- : int * float = (2, 3.)`
(j) `- : unit = ()`
(k) `- : int list = [7; 3; 2]`
(l) `- : 'a array list = [[| |]]`

Exercice 2 (Itératif, récursif, récursif terminale)

1. (a)

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

(b)

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

(c)

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

let suite_terminale n = suite_aux n 2.
;;
```

2. (a) On utilise le principe de la sentinelle :

```
let appartenance_iter t x =
  let n = Array.length t and i = ref 0 in
    let aux = t.(n-1) in
      t.(n-1) <-x;
      while x <> t.( !i) do
        i := !i+1
      done;
      ( !i < n-1) || (aux = x)
;;
```

(b)

```
let appartenance_rec l x = match l with
  | [] -> False
  | t::q -> (x = t) || (appartenance_rec q x)
;;
```

Exercice 3 (Les tris)

1. (a)

```
let insere i t =
  let k = ref (i-1) and aux = t.(i) in
    while (!k >= 0) && (t.(!k) > aux) do
      t.(!k+1) <- t.(!k);
      k := !k-1
    done;
    t.(!k+1) <- aux
;;
```

(b)

```
let tri_insertion t =
  let n = Array.length t in
    for i=1 to n-1 do
      insere i t
    done;
    t
;;
```

(c) — Complexité de `insere i t` :

La taille des données est i et les opérations fondamentales sont les comparaisons et les affectations. Notons $C_c(i)$ le nombre de comparaisons et $C_a(i)$ le nombre d'affectations.

A chaque itération, on fait 2 comparaisons et 1 affectation. Comme nous faisons i itérations, $C_c(i) = 2i$ et $C_a(i) = i+1$ (car on termine par l'affectation `t.(!k+1) <- aux`).

— Complexité de `tri_insertion t` :

La taille des données est la longueur n du tableau et les opérations fondamentales sont toujours les comparaisons et les affectations. Notons $T_c(n)$ le

nombre de comparaisons et $T_a(n)$ le nombre d'affectations. Alors :

$$\begin{aligned} T_c(n) &= \sum_{i=1}^{n-1} C_c(i) = 2\sum_{i=1}^{n-1} i = n(n-1) = O(n^2), \\ T_a(n) &= \sum_{i=1}^{n-1} C_a(i) = \sum_{i=1}^{n-1}(i+1) = \sum_{i=2}^{n} i = \frac{n(n+1)}{2} - 1 = O(n^2). \end{aligned}$$

2. (a)

```
let rec partition l = match l with
  | [] -> [] , []
  | [x] -> [x] , []
  | t1::t2::q -> let q1,q2 = partition q in (t1::q1 , t2::q2)
;;
```

(b)

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

(c)

```
let rec tri_fusion l = match l with
  | [] -> []
  | [x] -> [x]
  | _ -> let l1,l2 = partition l in
            fusion (tri_fusion l1) (tri_fusion l2)
;;
```

(d) Les opérations fondamentales sont les comparaisons.

— Pour la fonction `partition` :

Cette fonction ne nécessite aucune comparaison.

— Pour la fonction `fusion` :

La taille des données est la somme des longueurs n_1+n_2 des deux listes l_1 et l_2. Cette fonction nécessite au plus $n_1+n_2-1 = O(n_1+n_2)$ comparaisons.

— Pour la fonction `tri_fusion` :

La taille des données est la longueur n de la liste l. Notons $T(n)$ le nombre de comparaisons. On a alors :

$$T(n) = \underbrace{0}_{\text{partition}} + \underbrace{2T(n/2)}_{\text{appels récursifs}} + \underbrace{O(n)}_{\text{fusion}}$$

En appliquant le théorème maître de complexité, avec $q = 2$ et $\gamma = 1$, on obtient $T(n) = O(n\log_2(n))$.

Exercice 4

1.
```
let rec gcd a b =
  if b = 0 then
    a
  else
    if a >= b then
      gcd (a-b) b
    else
      gcd b a
;;
```
C'est un algorithme récursif terminal (il n'y a pas empilement puis dépilement d'appels récursifs).

2.
```
type frac = { num : int; denom : int } ;;
```

3.
```
let simp f =
  let b = abs f.denom in
    let a = f.num * (if f.denom < 0 then -1 else 1) in
      let p = gcd (abs a) b in
        { num = a/p ; denom = b/p }
;;
```

4. (a)
```
let add_frac f1 f2 =
  let a = f1.num * f2.denom + f2.num *f1.denom in
    let b = f1.denom * f2.denom in
      simp { num = a; denom = b}
;;
```
(b)
```
let neg_frac f = simp { num = -f.num ; denom = f.denom } ;;
```
(c)
```
let sub_frac f1 f2 = add_frac f1 (neg_frac f2) ;;
```
(d)
```
let mul_frac f1 f2 =
  let a = f1.num * f2.num and b = f1.denom * f2.denom in
    simp { num = a ; denom = b }
;;
```
(e)
```
let inv_frac f = simp { num = f.denom ; denom = f.num} ;;
```
(f)
```
let div_frac f1 f2 = mul_frac f1 (inv_frac f2) ;;
```