Correction du devoir du Vendredi 7 Mai

Exercice 1 (La suite de Fibonacci)

1. (a)

```
let fib1 n =
  let a = ref 0 and b = ref 1 and aux = ref 0 in
    for k = 1 to n do
      aux := !b;
      b := !a + !b;
      a := !aux
    done;
  !a
;;
```

(b)

```
let rec fib2 n = match n with
  | 0 -> 0
  | 1 -> 1
  | _ -> fib2 (n-1) + fib2 (n-2)
;;
```

(c)

```
(*fonction auxiliaire*)

let rec fib3_aux n acc1 acc2 = match n with
  | 0 -> acc1
  | 1 -> acc2
  | _ -> fib3_aux (n-1) acc2 (acc1+acc2)
;;

(*fonction chapeau*)

let fib3 n = fib3_aux n 0 1 ;;
```

(d)

```
let rec fib4_aux n table =
  if (table.(n) = -1) then
     table.(n) <- (fib4_aux (n-1) table) + (fib4_aux (n-2) table);
  table.(n)
;;

let fib4 n=
   let table = Array.make (n+1) -1 in
      table.(0) <- 0;
      table.(1) <- 1;
      fib4_aux n table
;;
```

(e) La taille des données est n et les opérations fondamentales sont les additions. Notons $C(n)$ le nombre d'addiction. On a alors :
— Pour `fib1`, la complexité $C_1(n) = O(n)$ est linéaire.

— Pour `fib2`, la complexité $C_2(n) = O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$ est exponentielle.
— Pour `fib3`, la complexité $C_3(n) = O(n)$ est linéaire.
— Pour `fib4`, la complexité $C_4(n) = O(n)$ est linéaire.

2. (a) Soit $p \in \mathbb{N}$. Montrons par récurrence sur $q \in \mathbb{N}^*$ la propriété

$$\mathcal{P}(q) : "F_{p+q} = F_{p+1}F_q + F_pF_{q-1}".$$

Ini. $F_{p+1} = 1 \times F_{p+1} + 0 \times F_p = F_1F_{p+1} + F_0F_p$ donc $\mathcal{P}(1)$ est vraie.
$F_{p+2} = 1 \times F_{p+1} + 1 \times F_p = F_2F_{p+1} + F_1F_p$ donc $\mathcal{P}(2)$ est vraie.
Héré. Soit $q \geqslant 2$. Supposons que $\mathcal{P}(q-1)$ et $\mathcal{P}(q)$ soient vraies.

$$\begin{aligned} F_{p+q+1} &= F_{p+q} + F_{p+q-1} \\ &= F_{p+1}F_q + F_pF_{q-1} + F_{p+1}F_{q-1} + F_pF_{q-2} \\ &= F_{p+1}(F_q + F_{q-1}) + F_p(F_{q-1} + F_{q-2}) \\ &= F_{p+1}F_{q+1} + F_pF_q. \end{aligned}$$

Donc $\mathcal{P}(q+1)$ est vraie.
Ccl. Par récurrence, pour tout $q \in \mathbb{N}^*$, $F_{p+q} = F_{p+1}F_q + F_pF_{q-1}$.

(b) En prenant $p = q = n$, on a :

$$F_{2n} = F_{n+1}F_n + F_nF_{n-1} = F_{n+1}F_n + F_n(F_{n+1} - F_n) = 2F_nF_{n+1} - F_n^2.$$

En prenant $p = n$ et $q = n+1$, on a :

$$F_{2n+1} = F_{n+1}^2 + F_n^2.$$

Enfin, en prenant $p = n$ et $q = n+2$, on a :

$$F_{2n+2} = F_{n+1}F_{n+2} + F_nF_{n+1} = F_{n+1}(F_{n+1} + F_n) + F_nF_{n+1} = F_{n+1}^2 + 2F_nF_{n+1}.$$

(c)

```
let rec fibo5 n = match n with
  | 0 -> 0 , 1
  | _ -> let a,b = fibo5 (n/2) in
    if n mod 2 = 0 then
      2*a*b - a*a , a*a + b*b
    else
      a*a + b*b , 2*a*b + b*b
;;
```

Si on souhaite une fonction donnant simplement la valeur de F_n, on peut créer une fonction chapeau comme ceci :

```
let fibo6 n = fst (fibo5 n) ;;
```

(d) Voici le théorème demandé (à connaitre par coeur !!) :

Soit $q \in \mathbb{N}^*$, $\gamma \in \mathbb{R}_+$ et l'équation de complexité :

$$C(n) = qC(n/2) + O(n^\gamma)$$

— Si $\log_2(q) = \gamma$, alors $C(n) = O(n^\gamma \log_2(n))$.
— Si $\log_2(q) > \gamma$, alors $C(n) = O(n^{\log_2(q)})$.
— Si $\log_2(q) < \gamma$, alors $C(n) = O(n^\gamma)$.

Ici, la taille des données est n et les opérations fondamentales sont les additions et les multiplications. Notons $C(n)$ le nombre d'additions et de multiplications. On a alors :

$$C(n) = C(n/2) + 7 = C(n/2) + O(1).$$

En appliquant le théorème maitre de complexité précédent, avec $q = 1$ et $\gamma = 0$, on obtient $C(n) = O(\log_2(n))$.

Exercice 2 (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 3 (Nombre d'occurrences d'un élément dans un ensemble)

1. (a)

```
let occurences1 a x =
  let n = Array.length a and compt = ref 0 in
    for k = 0 to (n-1) do
      if a.(k) = x then compt := !compt+1
    done;
  !compt
;;
```

(b) Proposons l'invariant de boucle : Soit $k \in [\![1, n]\!]$,

($\mathcal{H}_k$) "A la fin de la k-ième itération, *compt* contient le nombre de valeurs égales à x parmi $\{a.(0), \dots, a.(k-1)\}$."

• ($\mathcal{H}_1$) : A la première itération, on teste si $a.(0) = x$. Si c'est le cas, la variable *compt*, qui était initialisée à 0, prend la valeur $0 + 1 = 1$. Sinon, la variable *compt* reste égale à 0. Dans tous les cas, à la fin de la première itération, *compt* contient le nombre de valeurs égales à x parmi $\{a.(0)\}$.

• Soit $k \in [\![1, n]\!]$. Supposons ($\mathcal{H}_{k-1}$), c'est-à-dire qu'au début de la k-ième itération, *compt* contient le nombre de valeurs égales à x parmi $\{a.(0), \dots, a.(k-2)\}$. On teste alors si $a.(k-1) = x$. Si c'est le cas, la variable *compt* prend la valeur *compt* + 1. Sinon, la variable *compt* reste inchangée. Dans tous les cas, à la fin de la k-ième itération, *compt* contient bien le nombre de valeurs égales à x parmi $\{a.(0), \dots, a.(k-1)\}$. Donc ($\mathcal{H}_k$) est vraie.

• Par principe de récurrence, pour tout k appartenant à $[\![1, n]\!]$, ($\mathcal{H}_k$) est vraie. Notre algorithme est donc correct.

(c) On effectue n itérations avec à chaque fois une comparaison. On donc une complexité $C(n) = n = O(n)$ qui est linéaire.

2. (a)

```
let rec occurences2 a x = match a with
  | [] -> 0
  | t::q when (t=x) -> 1+(occurences2 q x)
  | t::q -> occurences2 q x
;;
```

(b) Nous proposons comme suite strictement décroissante la suite "longueur de la liste a". Si a est de longueur $n \in \mathbb{N}^*$, `occurences2 a x` amène un appel récursif sur la queue de a de longueur $n - 1 < n$.

L'ordre usuel sur $\mathbb{N}$ étant bien fondé, notre algorithme se termine.

(c) Choisissons comme taille de données n la longueur de la liste et comme opération fondamentale la comparaison. En notant $C(n)$ la complexité au pire, on obtient :

$$C(0) = 0 \text{ et } \forall n \in \mathbb{N}^*,\ C(n) = 1 + C(n-1).$$

Donc $C(n) = n = O(n)$. La complexité est linéaire.

3.

```
let occurrence3 a x =
  let rec occurences3 a x acc = match a with
    | [] -> acc
    | t::q when (t=x) -> occurences3 q x (acc+1)
    | t::q -> occurences3 q x acc
  in occurences3 a x 0
;;
```