Corrigé TD complexité

1.
```
let rec croit = function
  | [a] -> true
  | a::b::q -> (a <= b) && croit (b::q)
;;
let rec decroit = function
  | [a] -> true
  | a::b::q -> (a >= b) && decroit (b::q)
;;
let monotone l = match l with
  [] -> true
  | [a] -> true
  | a::b::q -> if a<=b then croit l else decroit l
;;
```

2.
   - La fonction `aux` parcourt la liste en incrémentant la case du tableau dont on a trouvé une occurrence.

     ```
     let compte m l =
      let t = Array.make m 0 in
      let rec aux = function
           [] -> ()
       | k::q -> t.(k) <- t.(k) + 1; aux q
      in aux l;
      t ;;
     ```

   - La fonction récursive `aux` à l'intérieur de la fonction `tri` est telle que `aux i` remplit la liste triée en parcourant le tableau t jusqu'à la case i.

     ```
     (* add a k l ajoute k occurrences de a devant la liste l *)
     let rec add a k l = match k with
       0 -> l
      | _ -> a::(add a (k-1) l)
     ;;
     let tri m l =
      let t = compte m l in
      let rec aux i =
        if i = m then []
        else add i t.(i) (aux (i+1))
      in aux 0 ;;
     ```

   - La fonction `compte` crée un tableau de taille m et parcourt la liste. Le coût de ce tri est un $\Theta(n+m)$ pour une liste de longueur n. Le résultat obtenu peut sembler meilleur que la borne théorique $n \log n$ mais il s'appuie sur l'hypothèse supplémentaire que toutes les données sont comprises entre 0 et $m-1$, donc le problème est d'une autre nature.

3.
   - Avec `l=[1;2;5;10;12]` et $s = 15$, l'algorithme glouton retourne `[12;2;1]` alors que la solution minimale en nombre de pièces est `[10;5]`.

   -
     ```
     let rendu l s =
       let rec aux l s = match l with
           [] -> if s=0 then [] else failwith "impossible"
         | t::q when t<=s -> t::(aux l (s-t))
         | t::q -> aux q s
       in aux (List.rev l) s ;;
     ```

4.
```
let rec valu p = match p with
 [] -> failwith "erreur"
 | (a,m) :: q when a <> 0 -> m
 | (a,m) :: q -> valu q
;;
let rec simplifie = function
  [] -> []
 | (a,k) :: q when a = 0 -> simplifie q
 | (a,k) :: q -> (a,k) :: (simplifie q)
;;
let deg p =
 let rec deg_aux p = match p with
  [] -> failwith "erreur"
  | [(a,n)] -> n
  | _ :: q -> deg_aux q
 in deg_aux (simplifie p)
;;
let rec affiche = function
    [] -> ""
 | [a,0] -> "a"
 | [a,n] when a=1 ->  "X" ^ (if n=1 then "" else "^" ^ string_of_int n)
 | [a,n] when a= -1 -> "-X" ^ (if n=1 then "" else "^" ^ string_of_int n)
 | [a,n] -> (string_of_int a) ^ "X" ^ (if n=1 then "" else "^" ^  string_of_int n)
 | (a,0)::q when a=1 -> (affiche q) ^ "+" ^ (string_of_int a)
 | (a,0)::q when a<0 -> (affiche q) ^ (string_of_int a)
 | (a,0)::q -> (affiche q) ^ "+" ^ (string_of_int a)
 | (a,n)::q when a=1 -> (affiche q) ^ "+" ^ "X" ^ (if n=1 then "" else "^" ^  string_of_int n)
 | (a,n)::q when a= -1 -> (affiche q) ^ "-" ^ "X" ^ (if n=1 then "" else "^" ^  string_of_int n)
 | (a,n)::q when a<0 -> (affiche q) ^ (string_of_int a) ^ "X" ^
                        (if n=1 then "" else "^" ^  string_of_int n)
 | (a,n)::q  -> (affiche q) ^ "+" ^ (string_of_int a) ^ "X" ^
               (if n=1 then "" else "^" ^  string_of_int n)
;;
let somme p1 p2 =
  let rec somme_aux p1 p2 = match (p1,p2) with
      ([],_) -> p2
    | (_,[]) -> p1
    | ((a1,d1)::q1 , (a2,d2)::q2) when d1 = d2 -> (a1+a2,d1) :: (somme_aux q1 q2)
    | ((a1,d1)::q1 , (a2,d2)::q2) when d1 < d2 -> (a1,d1) :: (somme_aux q1 p2)
    | ((a1,d1)::q1 , (a2,d2)::q2) -> (a2,d2) :: (somme_aux q2 p1)
  in simplifie (somme_aux p1 p2)
;;
let rec produit_monome p (a,k) = match p with
  [] -> []
 | (b,d)::q -> (b*a,d+k) :: produit_monome q (a,k)
;;
let produit p1 p2 =
  let rec produit_aux p1 p2 = match p1 with
     [] -> []
   | (a,d)::q -> somme (produit_monome p2 (a,d)) (produit_aux q p2)
  in simplifie (produit_aux p1 p2)
;;
let evalue p x =
 let d = deg p in let puiss = Array.make (d+1) 1 in
 for k = 1 to d do puiss.(k) <- x * puiss.(k-1) done;
 let rec aux = function
    [] -> 0
  | (a,k)::q -> a * puiss.(k) + aux q
 in aux p
```

```
;;
let rec dominant = function
  [a,d] -> (a,d)
 | t::q -> dominant q
;;
let rec factorise p a = match p with
 [] -> []
 | _ -> let (c,d) = dominant p in let q = simplifie(somme p [(a*c,d-1);(-c,d)])
        in somme (factorise q a) [c,d-1]
;;
```

5. Dans la fonction `longueur_max`, dans la boucle `for`, la référence `lg` indique la longueur maximale d'une suite croissante commençant à l'indice i. La référence `long` indique la longueur maximale d'une suite croissante commençant à un indice $\leqslant i$.

   Dans la fonction `plsc`, on mémorise en plus l'indice de départ `dep` d'une plus longue suite croissante.

```
let longueur_max t =
  let n = Array.length t and long = ref 1 in
  for i = 0 to n-1 do
    let lg = ref 1 and j = ref i in
while !j < n-1 && t.(!j) <= t.(!j +1) do
  (incr j; incr lg)
done;
if !lg > !long then long := !lg
  done;
  !long
;;
let plsc t =
  let n = Array.length t and long = ref 1 and dep = ref 0 in
  for i = 0 to n-1 do
    let lg = ref 1 and j = ref i in
while !j < n-1 && t.(!j) <= t.(!j +1) do
  (incr j; incr lg)
done;
if !lg > !long then (long := !lg; dep := i)
  done;
  Array.sub t !dep !long
;;
```