Correction du concours blanc

Option informatique, première année

Julien Reichert

Exercice 1

```
let rec croissante l = match l with
| [] | [_] -> true
| a::b::q -> a <= b && croissante (b::q);;
```

Exercice 2

```
let croissant tab =
  let n = Array.length tab in
  if n < 2 then true else
  (let i = ref 0 in
    while !i < n-1 && tab.(!i+1) >= tab.(!i) do incr i done;
    !i = n-1);;
```

Exercice 3

```
let rec inferieurs l x = match l with
| [] -> 0
| a::q -> inferieurs q x + (if a < x then 1 else 0);;
```

Exercice 4

```
let inferieurs_tab tab x =
  let rep = ref 0 in
    for i = 0 to Array.length tab - 1 do
      if tab.(i) < x then incr rep
    done; !rep;;
```

Exercice 5

Voir le cours pour l'énoncé. Dans le cadre de la dichotomie, la relation $c_n = c_{\frac{n}{2}} + \Theta(1)$ donne $c_n = \Theta(\log n)$.

Exercice 6

```
let rec fusion l ll = match (l, ll) with
| [], l2 -> l2
| l1, [] -> l1
| (a::q1), (b::q2) when a < b -> a::(fusion q1 (b::q2))
| l1, (b::q2) -> b::(fusion l1 q2);;

let rec fission = function
| [] -> [], []
| [a] -> [a], []
| a::b::q -> let l1, l2 = fission q in a::l1, b::l2;;

let rec tri_fusion l =
  if List.length l < 2 then l
  else let l1, l2 = fission l in fusion (tri_fusion l1) (tri_fusion l2);;
```

Exercice 7

```
let fusion_tab tab1 tab2 =
  let n1 = Array.length tab1 and n2 = Array.length tab2 in
    if n1 = 0 then tab2
    else if n2 = 0 then tab1
    else (
      let tab = Array.make (n1+n2) tab1.(0) (* Ici l'astuce nécessitant la précaution (ou un try) *) in
        let i1 = ref 0 and i2 = ref 0 in
          while !i1 < n1 && !i2 < n2 do
            if tab1.(!i1) < tab2.(!i2) then
              (tab.(!i1 + !i2) <- tab1.(!i1);
                incr i1)
            else
              (tab.(!i1 + !i2) <- tab2.(!i2);
                incr i2)
          done;
          while !i1 < n1 do
            tab.(!i1 + !i2) <- tab1.(!i1);
            incr i1
          done;
          while !i2 < n2 do
            tab.(!i1 + !i2) <- tab2.(!i2);
            incr i2
          done;
          tab
    );;

let rec tri_fusion_tab tab =
  let n = Array.length tab in
    if n < 2 then tab
    else
      let gauche = Array.sub tab 0 (n/2)
      and droite = Array.sub tab (n/2) (n-n/2) in
        fusion_tab (tri_fusion_tab gauche) (tri_fusion_tab droite);;
```

Exercice 8

On utilise le master theorem, en notant que le coût de la fission et celui de la fusion sont linéaires car on fait simplement un traitement de chaque élément, la relation $c_n = 2c_{\frac{n}{2}} + \Theta(n)$ donne $c_n = \Theta(n \log n)$.

Exercice 9

```
let rec nombre_inversions l = match l with
| [] | [_] -> 0
| a::q -> inferieurs q a + nombre_inversions q;;
```

Exercice 10

```
let nombre_inversions_tab tab =
  let rep = ref 0 in
    for i = 0 to Array.length tab - 1 do
      for j = i+1 to Array.length tab - 1 do
        if tab.(i) > tab.(j) then incr rep
      done done; !rep;;
```

Ici, on n'a pas réutilisé la fonction de la première section car il aurait fallu l'appeler sur un sous-tableau, entraînant un coût en espace non désiré, ou la réécrire pour s'autoriser à démarrer à l'indice qu'on souhaite, ce qui prend plus de temps sur une copie.

Exercice 11

En admettant un algorithme naïf pour déterminer le nombre d'éléments supérieurs entre la gauche et la droite, on doit faire un parcours des deux listes, donc une complexité quadratique au premier découpage. Toute l'optimisation est alors ruinée.

Exercice 12

```
let decoupe l = let n = List.length l in (* complexité linéaire *)
    let rec extrait i ll = match i, ll with
    | 0, _ -> [], ll
    | _, [] -> failwith "Impossible"
    | _, a::q -> let l1, l2 = extrait (i-1) q in a::l1, l2
    in extrait (n/2) l;;
```

Au calcul de la taille `n` de la liste s'ajoutent $\frac{n}{2}$ (à l'arrondi près) appels récursifs, pour un coût total linéaire.

Exercice 13

L'astuce revient à trier à la volée les listes lors du calcul du nombre d'inversions, de sorte qu'on sache exactement combien d'éléments de la liste de gauche sont supérieurs à l'élément de la liste de droite considérés : le nombre d'éléments restants dans la liste, en admettant celle-ci triée (l'autre le sera aussi, mais ce n'est pas nécessaire pour la preuve).

Version listes :

On écrit une fonction auxiliaire qui fusionne deux listes triées en comptant le nombre de supériorités entre la première et la deuxième. Comme on ne peut pas se permettre de calculer à chaque fois des tailles de liste, il faut trouver une astuce, soit compter dans l'autre sens, soit calculer la taille au début et suivre à l'aide d'une variable les mises à jour. On n'oubliera cependant pas les éventuels ajouts finaux.

```
let fusionne_et_compte ll1 ll2 =
  let rec aux l1 l2 d = match (l1, l2) with
    | ([], _) -> 0, l2
    | (_, []) -> (d * List.length l1) , l1
    | (a::q1, b::q2) ->
      if a <= b then
        let nb, l = aux q1 l2 d in nb+d, a::l
      else
        let nb, l = aux l1 q2 (d+1) in nb, b::l
  in aux ll1 ll2 0;;

(* Version improvisée lors de la correction en classe, peut-être plus lisible *)
let fusionne_et_compte2 l1 l2 =
  let n1 = ref (List.length l1) in
    let rec aux ninv buff ll1 ll2 = match ll1, ll2 with
      |[],[] -> ninv, List.rev buff
      |a::q1, l::q2 -> if a < l then (decr n1 ; aux ninv (a::buff) q1 ll2)
        else aux (ninv + !n1) (l::buff) ll1 q2
      |[], _ -> ninv, (List.rev buff) @ ll2
      |_, [] -> ninv, (List.rev buff) @ ll1
    in aux 0 [] l1 l2;;

let nombre_inversions_dpr l =
  let rec aux liste = match liste with
    | [] | [_] -> 0, liste
    | _ -> let l1, l2 = decoupe liste in
      let n1, ll1 = aux l1
      and n2, ll2 = aux l2 in
      let n3, ll = fusionne_et_compte ll1 ll2 in (n1 + n2 + n3), ll
  in aux l;;
```

Exercice 14

Cette fois-ci, le coût qui aurait été obtenu par une double boucle devient linéaire, et avec le coût également linéaire du découpage le master theorem s'applique avec la formule de récurrence $c_n = 2c_{\frac{n}{2}} + \Theta(n)$, d'où là aussi $c_n = \Theta(n \log n)$.