Correction du DS 1

Julien Reichert

La correction des questions de cours étant dans le cours, elle ne sera pas donnée ici...

Exercices

Exercice 1 :

```
let premier_libre_depuis tab n =
  let taille = Array.length tab in
  let apparait_decale = Array.make taille false in
  for i = 0 to taille - 1 do
    if tab.(i) >= n && tab.(i) < n + taille then apparait_decale.(tab.(i) - n) <- true
  done;
  let j = ref 0 in
  while !j < taille && apparait_decale.(!j) do incr j done;
  !j + n;;
```

Sous peu, la structure de dictionnaire rendra la résolution plus intuitive. Ici, on va déterminer pour toutes les valeurs potentielles si elles ont été attribuées, en consommant un espace mémoire linéaire pour éviter un temps quadratique, la complexité sera linéaire car le corps de chaque boucle s'exécute en temps constant, la deuxième boucle ayant comme variant `taille - !j`.

En ce qui concerne la preuve de correction, on peut utiliser comme invariant de la première boucle « Après le tour pour `i`, à tout indice `j` de `apparait_decale` il y a un `true` si, et seulement si, on a déjà rencontré au moins une fois `j+n`. », la deuxième boucle étant simplement la recherche du premier `false` d'un tableau de booléen, correspondant à un nombre absent de `tab` à l'ajout de `n` près. On notera que si par hasard le tableau contenait tous les nombres de `n` à `n + taille - 1`, la boucle conditionnelle s'arrêterait pour la première des deux conditions conjointes et la valeur retournée serait, comme il faudrait s'y attendre, `n + taille - 1`.

Exercice 2 :

```
let dichotomie tab n =
  let taille = Array.length tab in
  let debut = ref 0 in
  let fin = ref (taille - 1) in
  while !debut < !fin do
    let milieu = (!debut + !fin) / 2 in
    if tab.(milieu) = n then
      ( debut := milieu; fin := milieu )
    else if tab.(milieu) < n then
      debut := milieu + 1
    else
      fin := milieu
  done;
  !debut;;
```

```
let dichotomie_decalage tab ind =
  let taille = Array.length tab in
  let debut = ref (ind + 1) in
  let fin = ref taille in
  while !debut < !fin do
    let milieu = (!debut + !fin) / 2 in
    if tab.(milieu) - milieu = tab.(ind) - ind then
      debut := milieu + 1
    else
      fin := milieu
  done;
  !debut-1;;

let premier_libre_depuis tab n =
  let ind = dichotomie tab n in
  if tab.(ind) <> n then n
  else
    let ind2 = dichotomie_decalage tab ind in
    tab.(ind2) + 1;;
```

Ici, il s'agit de localiser la valeur `n` par dichotomie, si elle est dans le tableau. Dans le cas contraire, la réponse de la fonction est immédiate. Sinon, une deuxième dichotomie permet de regarder le premier indice ultérieur du tableau où l'on n'a pas avancé d'un cran exactement.

Les variants des deux boucles conditionnelles sont la partie entière du logarithme en base deux de `!fin - !debut` qu'on ne calcule pas lorsque `!debut` et `!fin` sont égaux, ce qui signifie de toute manière que l'on arrête la boucle. Cette valeur est bien un entier positif qui décroît strictement à chaque tour de boucle car l'écart entre `!debut` et `!fin` est au pire divisé par deux à chaque tour de boucle (si l'écart est pair, il est vraiment divisé par deux, et s'il est impair il est divisé par deux en arrondissant par défaut). On en déduit la complexité logarithmique des deux fonctions et donc de la fonction principale qui les appelle en séquence, **ce qui additionne les complexités**.

Quant à la preuve de correction, elle repose sur l'invariant de boucle suivant pour la première dichotomie : « Si `n` est dans le tableau, alors il y est entre la position `!debut` et la position `!fin` inclus. », ce qui permet de conclure lorsque les deux indices se sont rejoints pour ce qui est de la présence de `n`. Si `n` est présent, la première valeur absente est un de plus que le dernier indice où l'écart entre valeur et indice est identique à cet écart évalué là où on a trouvé `n`. L'invariant de boucle de la deuxième dichotomie est alors « Les nombres entre l'indice `ind` et l'indice `!debut-1` sont consécutifs, mais cela n'est pas vrai jusqu'à l'indice `!fin` (qui déborde potentiellement). », dont la preuve d'hérédité n'est pas évidente car elle repose sur la stricte croissance du tableau et en quelque sorte le principe des tiroirs.

Exercice 3 :

Ce n'est peut-être pas encore le moment d'optimiser le calcul du maximum et du minimum pour multiplier la complexité par trois quarts...

Ici, on considère une liste de flottants, ce qui ne se voit que dans le calcul final.

```
let moyminmax l =
  let rec minmax l = match l with
  | [] -> failwith "Liste vide"
  | [a] -> a, a
  | a::q -> let mini, maxi = minmax q in min a mini, max a maxi
  in let mini, maxi = minmax l in (mini +. maxi) /. 2.;;
```

Exercice 4 :

```
let list_of_array tab =
  let rec mouline i =
    if i = Array.length tab then []
    else tab.(i) :: mouline (i+1)
  in mouline 0;;

let array_of_list l =
  let taille = List.length l in
  if taille = 0 then [||] (* Sinon comment initialiser ? *)
  else let rep = Array.make taille (List.hd l) in
  let rec mouline i l = match l with
  | [] -> ()
  | a::q -> rep.(i) <- a; mouline (i+1) q
  in mouline 0 l; rep;;
```

Exercice 5 :

```
let rec nthnth l i j = match l, i, j with
| [], _, _ -> failwith "Liste de listes trop courte"
| []::_, 0, _ -> failwith "Liste intérieure trop courte"
| (a::_)::_, 0, 0 -> a
| (_::q)::_, 0, _ -> nthnth [q] 0 (j-1) (* Les autres listes n'ont plus d'importance en pratique. *)
| _::q, _, _ -> nthnth q (i-1) j;;
```

Exercice 6 :

```
let slice tab d f p =
  let n = Array.length tab in
  if p = 0 then failwith "Le pas est nul !";
  if d < 0 then failwith "L'indice de début est négatif !";
  if d >= n then failwith "L'indice de début est trop grand !";
  if f < -1 then failwith "L'indice de fin est trop petit !";
  if f > n then failwith "L'indice de fin est trop grand";
  let taille = ref 0 in
  let ind = ref d in
  while (p > 0 && !ind < f) || (p < 0 && !ind > f) do (* parenthèses pour la lisibilité *)
    incr taille;
    ind := !ind + p
  done;
  let rep = Array.make !taille tab.(d) in
  for i = 0 to !taille - 1 do
    rep.(i) <- tab.(d + p * i)
  done;
  rep;;
```

Problème 1

Question P1 :

Une valeur nulle a des zéros partout dans l'exposant et la mantisse, et un signe au choix pour simuler 0^+ et 0^-.

```
let zero_b tab =
  let i = ref 1 in
  while !i < 64 && not tab.(!i) do incr i done;
  !i = 64;;
```

Question P2 :

Ici, la vérification porte sur l'exposant, devant avoir des zéros partout ou des uns partout.

```
let denor_b tab =
  let i = ref 2 in
  while !i < 12 && tab.(!i) = tab.(1) do incr i done;
  !i = 12;;
```

Question P3 :

Le plus petit nombre strictement positif représentable en virgule flottante (hors valeurs exceptionnelles) sur 64 bits s'écrit 0 00000000001 $0\ldots0$ et correspond à 2^{-1022} donc environ 10^{-300}.

Le plus grand nombre s'écrit 0 11111111110 $1\ldots1$ et correspond à $2^{1023} \times (1 + 2^{-1} + 2^{-2} + \cdots + 2^{-52}) = 2^{1023} \times (2 - 2^{-52})$ qui est de l'ordre de 10^{300}.

Le produit des deux fait donc $2 \times (2 - 2^{-52}) = 4 - 2^{-51}$.

Question P4 :

Si le premier bit diffère, le signe permet de trancher. Sinon, de manière assez intéressante, la priorité étant à l'exposant puis à la mantisse, la première différence désigne quel flottant est le plus grand. Attention cependant, si les deux sont négatifs, l'ordre n'est pas le même !

```
exception Superieur;;
exception Inferieur;;

let sup_b tab1 tab2 =
  try
    match tab1.(0), tab2.(0) with
    | false, true -> true
    | true, false -> false
    | true, true ->
      for i = 1 to 63 do
        if tab1.(i) <> tab2.(i) then
          if tab1.(i) then raise Inferieur else raise Superieur
      done; true (* car égal *)
    | _ ->
      for i = 1 to 63 do
        if tab1.(i) <> tab2.(i) then
          if tab2.(i) then raise Inferieur else raise Superieur
      done; true (* car égal *)
  with
  | Superieur -> true
  | Inferieur -> false;;
```

Question P5 :

Attention, notre flottant peut être supérieur ou inférieur à 1. Version de complexité linéaire en la réponse :

```
let exposant x =
  let rep = ref 0 in
  let d_p_rep = ref 1. in
  if x >= 1. then
  begin
    while !d_p_rep <= x do
      incr rep;
      d_p_rep := !d_p_rep *. 2.
    done; !rep - 1
  end
  else
  begin
    while !d_p_rep > x do
      decr rep;
      d_p_rep := !d_p_rep /. 2.
    done; !rep
  end;;
```

Question P6 :

```
let intbin n =
  if n < -1022 || n > 1023 then
    failwith "Exposant interdit !";
  let res = Array.make 11 false in
  let np1023 = ref (n + 1023) in
  let ind = ref 10 in
  while !np1023 <> 0 do
    if !np1023 mod 2 = 1 then
      res.(!ind) <- true;
    decr ind;
    np1023 := !np1023 / 2
  done;
  res;;
```

Question P7 :

```
let mantisse f =
  if f < 0. || f >= 1. then
    failwith "Valeur en-dehors de l'intervalle [0;1[ !";
  let res = Array.make 52 false in
  let fref = ref f in
  for i = 0 to 51 do
    fref := 2. *. !fref;
    if !fref > 1. then
      (res.(i) <- true;
      fref := !fref -. 1.)
  done;
  (res, !fref > 0.5 || !fref = 0.5 && res.(51));;
```

La règle exacte pour l'arrondi final peut faire l'objet de tolérance lors de la correction. Il s'agit d'arrondir au plus proche, et vers la valeur paire en cas d'équidistance, donc on met dans ce dernier cas une retenue si, et seulement si, le bit final est à 1, d'où la formule.

Question P8 :

```
let representation f =
  let reponse = Array.make 64 false in
  let absf = (if f < 0. then (reponse.(0) <- true; -. f) else f) in
  let n = ref (exposant absf) in
  let soustabd, retenue = mantisse (absf /. 2. ** (float_of_int !n) -. 1.) in
  if retenue then
  begin
    let j = ref 51 in
    while !j >= 0 && soustabd.(!j) do
      soustabd.(!j) <- false;
      decr j
    done;
    if !j >= 0 then soustabd.(!j) <- true
    else incr n
  end;
  let soustabg = intbin !n in
  for i = 0 to 10 do
    reponse.(i+1) <- soustabg.(i)
  done;
  for i = 0 to 51 do
    reponse.(i+12) <- soustabd.(i)
  done;
  reponse;;
```

Problème 2

Pour les personnes intéressées, le numéro du puzzle sur le site Robozzle est précisé à chaque fois, ceci permet en particulier de tester la solution.

La représentation des solutions est intuitive et correspond aussi au site.

Premier puzzle (numéro 24)

F1 : ↑↷ F2
F2 : ↑↶ F1

Deuxième puzzle (numéro 39)

F1 : F3 F2 F3 F3 F2 F2 F3 F2 F2 F3
F2 : ↑↑↷
F3 : ↑↑↶

On peut finir F1 par F2, le tout est de ne pas oublier de lancer une fonction qui avance encore. L'avantage est que le nombre d'instructions allouées permet d'éviter cette omission.

Troisième puzzle (numéro 539)

F1 : ↑ ↶ F2 F1
F2 : ↶↶ F3
F3 : ↑ ↷ F3

D'après le site, il existe une solution avec sept instructions.

Quatrième puzzle (numéro 648)

Une fonction est inutile, et dans l'idée on se sert de trois fonctions pour simuler une fonction avec trois instructions plus un appel récursif à la fonction elle-même :

F1 : ↶ F2
F2 : ↑ F3
F3 : ↶ F4
F4 : ↶ F2

Cinquième puzzle (numéro 587)

Certaines instructions sont automatiques, ensuite il reste à broder :

F1 : ↶ ↑ F2 F1
F2 : ↑↷↷↑

Autre solution :

F1 : ↑ ↶ F2 F1
F2 : ↑ ↷↷ F2

D'après le site, il existe une solution avec sept instructions.

Sixième puzzle (numéro 656)

C'était le puzzle le plus compliqué. En pratique, les colonnes étaient de taille si chaotique qu'on devait se douter que ces tailles n'avaient pas d'importance (peut-être y a-t-il un message codé, ce serait amusant !). Ainsi, il fallait empiler des instructions d'avancement mises en attente par des appels récursifs prioritaires. À l'instar du premier exercice du TD0, la question qu'on peut se poser est « Comment passer de la configuration de départ à la même configuration mais un cran en avant et en ayant visité la colonne ? », et la réponse est par la fonction F1 dont la partie concernant la colonne se fait dans F2, au rétablissement de l'orientation près.

F1 : ↶ F2 ↶↑ F1
F2 : ↑ F2 ↶↶ ↑

L'ordre des trois instructions du milieu de F2 n'est pas important.

D'après le site, il existe une solution avec neuf instructions.