Correction du DS 4

Julien Reichert

La correction des questions de cours étant dans le cours, elle ne sera pas donnée ici. De même, les exercices issus des TD et TP ont leur correction déjà publiée.

Exercices

Exercice 1 : [Version intuitive en temps quadratique, certains ayant proposé une solution en temps linéaire...]

```
bool tous_extremaux(int* tab, int n)
{
  for (int i = 0 ; i < n - 1 ; i += 1)
  {
    if (tab[i] == tab[i+1]) return false;
    bool sup = tab[i] > tab[i+1];
    for (int j = i+1 ; j < n ; j += 1)
    {
      if (tab[j] >= tab[i] && sup || tab[j] <= tab[i] && !sup) return false;
    }
  }
  return true;
}
```

Exercice 2 :

```
let combien_ne_se_trompent_pas seuils identites =
  let n = Array.length seuils in let reponse = ref 0 in
  for i = 0 to Array.length identites - 1 do
    let (prenom, nom) = identites.(i) in let init_p = prenom.[0] in let init_n = nom.[0] in
    let indice = ref 0 in
      while !indice < n do
        if init_p <= seuils.[!indice] && init_n <= seuils.[!indice] then incr reponse;
        if init_p <= seuils.[!indice] || init_n <= seuils.[!indice] then indice := n
        else incr indice
      done;
    !reponse
```

On quitte la boucle quand le seuil n'est plus atteint par une initiale (on incrémente la réponse si c'est par les deux).

Problème 1

Question P1.1 :

On a successivement les termes $19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1$.

Question P1.2 :

Le temps de vol en altitude est 6 et l'altitude maximale est 88. Si u_0 est pair, le temps de vol est nécessairement 1 car $u_1 < u_0$, étant sa moitié. Une fois que $u_n = 1$, la suite boucle sur les trois valeurs $1, 4, 2$ dans cet ordre.

Question P1.3 :

```
int terme_suivant(int un)
{
  return un % 2 == 0 ? un / 2 : 3 * un + 1;
}
```

Question P1.4 :

```
int temps_vol_altitude(int n)
{
  int un = n;
  int indice = 0;
  while (un >= n)
  {
    un = terme_suivant(un);
    indice += 1;
  }
  return indice;
}
```

Question P1.5 :

On utilise un tableau en guise de structure de mémoire. La mémoïzation sera simplement partielle, pour ne pas avoir besoin d'une quantité inconnue de mémoire a priori.

```
int recursion_ttv(int* memoire, int n, int k)
{
  if (k > n) return 1 + recursion_ttv(terme_suivant(k));
  if (memoire[k] == -1) memoire[k] = 1 + recursion_ttv(terme_suivant(k));
  return memoire[k]
}

int* tous_temps_vol(int n)
{
  int* reponse = malloc((n+1) * sizeof(int));
  for (int i = 0 ; i <= n ; i += 1) reponse[i] = -1;
  reponse[1] = 0;
  for (int i = 2 ; i <= n ; i += 1) reponse[i] = recursion_ttv(memoire, n, i);
  return reponse;
}
```

On en profite pour insister sur le fait que le standard du langage C ne permet pas de définir de fonction locale.

Question P1.6 :

Pas une grande différence dans le fonctionnement, seule la formule change.

```
int recursion_tam(int* memoire, int n, int k)
{
  if (k > n)
  {
    int rec = recursion_tam(terme_suivant(k));
    return rec > k ? rec : k;
  }
  if (memoire[k] == -1)
  {
    int rec = recursion_tam(terme_suivant(k));
    memoire[k] = rec > k ? rec : k;
  }
  return memoire[k];
}

int* toutes_altitudes_maximales(int n)
{
  int* reponse = malloc((n+1) * sizeof(int));
  for (int i = 0 ; i <= n ; i += 1) reponse[i] = -1;
  reponse[1] = 1;
  for (int i = 2 ; i <= n ; i += 1) reponse[i] = recursion_tam(memoire, n, i);
  return reponse;
}
```

Problème 2

Question P2.1 :

Une séquence finie non vide possède un minimum global, qui est a fortiori local.

Soit une séquence infinie dont les éléments sont pris dans un ensemble muni d'un ordre total bien fondé.

On considère un élément arbitraire x_0 de cette séquence. Alors de deux choses l'une : soit l'élément est un minimum local, soit il a un voisin strictrement inférieur. On crée alors une suite (x_n) telle que x_{n+1} soit le plus petit nombre entre x_n et ses voisins, en faisant abstraction des positions permettant de trouver ces éléments.

La suite (x_n) ne peut pas à la fois être infinie et strictement décroissante, or elle est clairement décroissante et dès que deux éléments consécutifs x_n et x_{n+1} sont identiques, c'est que x_n n'avait aucun voisin strictement inférieur, ce qui veut dire qu'on était en présence d'un minimum local.

Ce principe est exactement celui qu'on utilise dans la dernière question.

Question P2.2 :

```
let indice_min_local tab =
  let n = Array.length tab in
    if n = 0 then failwith "Tableau vide"
    else if n = 1 || tab.(0) <= tab.(1) then 0
    else
      let i = ref 1 in
        while !i < n-1 && tab.(!i) > tab.(!i+1) do
          incr i
        done;
      !i
```

La complexité au pire des cas est linéaire en la taille du tableau.

Question P2.3 :

On fait une dichotomie : soit l'élément du milieu est un minimum local, soit il a un voisin strictement inférieur, et on part dans le demi-tableau contenant ce voisin, car il y a au moins un minimum local dans ce demi-tableau, que les éléments soient strictement décroissants du milieu au bord (qui est un minimum local) ou que cette décroissance s'arrête, donc sur un minimum local.

Pour faire bonne mesure, tester les éléments du bord au début procure une probabilité assez forte de tomber tout de suite sur un minimum local, puisque ces éléments n'ont qu'un voisin.

Mieux encore, cela garantit qu'on ne débordera pas lors de la dichotomie (même si la condition d'arrêt suffit, on n'a vraiment pas besoin de se poser de question).

```
let indice_min_local_dicho tab =
  let n = Array.length tab in
    if n = 0 then failwith "Tableau vide"
    else if n = 1 || tab.(0) <= tab.(1) then 0
    else if tab.(n-1) <= tab.(n-2) then n-1
    else
      let ind_min = ref 0 and ind_max = ref (n-1) in
        while !ind_min < !ind_max do
          let ind_mil = (!ind_min + !ind_max)/2 in
            if tab.(ind_mil) <= tab.(ind_mil-1) && tab.(ind_mil) <= tab.(ind_mil+1)
              then begin ind_min := ind_mil; ind_max := ind_mil end (* sortie de boucle manuelle *)
            else if tab.(ind_mil) > tab.(ind_mil-1)
              then ind_max := ind_mil
            else
              ind_min := ind_mil
        done;
      !ind_min
```

Question P2.4 :

La complexité ne peut pas être logarithmique dans la mesure où l'accès au dernier élément nécessite de parcourir la liste en entier, et si la liste est décroissante le dernier élément est le seul minimum local. En particulier, si on avait converti la liste en tableau pour faire une dichotomie, le cout de recopiage linéaire aurait été plus important que la recherche proprement dite.

```
let min_local_liste l =
  match l with [] -> failwith "Liste vide"
  | tete::queue ->
    let rec aux pos elt_prec liste = match liste with
    | [] -> elt_prec
    | elt::q when elt_prec <= elt -> elt_prec (* Finit plus vite qu'avec < *)
    | elt::q -> aux (pos+1) elt q
    in aux 0 tete queue;;
```

Question P2.5 :

Il est bien entendu possible de procéder par pas de taille la précision souhaitée, mais la dichotomie est évidemment mieux adaptée. La fonction peut être itérative (auquel cas on peut se référer à la recherche de zéro par dichotomie en Python, une transcription adaptée à notre problème s'écrit aisément) ou récursive, comme ci-dessous.

Attention, ce qui compte ici est d'étudier les variations sur une petite partie de l'intervalle pour détecter la présence d'une zone où la fonction décroît puis d'une zone où elle croît, puis de restreindre la taille de l'intervalle englobant ces deux zones.

```
let rec minlocalfonction f a b eps =
    let mil = (a +. b) /. 2 in
    if b -. a < 2. *. eps then mil
    else if f(a) <= f(a +. eps) then a (* il y a un minimum global dans la zone *)
    else if f(b) <= f(b -. eps) then b (* parce qu'à un moment ça descend entre près du bord et le bord *)
    else if f(mil) <= f(mil +. eps) && f(mil) <= f(mil -. eps) then mil
    else if f(mil) > f(mil +. eps) then minlocalfonction f mil b eps
    else minlocalfonction f a mil eps;;
```

Question P2.6 :

On peut simplement adapter la fonction sur les tableaux, en notant que rien n'impose que les lignes soient de même taille. Ici, on va faire un parcours avec rattrapage d'exceptions pour la flexibilité.

Vu qu'on va modulariser ultérieurement, autant écrire tout de suite une fonction annexe utile déterminant si une position correspond à un minimum local.

```
let est_min_local tab i j =
let n = Array.length tab in
  (i = 0 || j >= Array.length tab.(i-1) || tab.(i).(j) <= tab.(i-1).(j))
  && (i = n-1 || j >= Array.length tab.(i+1) || tab.(i).(j) <= tab.(i+1).(j))
  && (j = 0 || tab.(i).(j) <= tab.(i).(j-1))
  && (j = Array.length tab.(i)-1 || tab.(i).(j) <= tab.(i).(j+1));;

exception Trouve of int * int;;

let indice_min_local_tabtab tab =
  let n = Array.length tab in
  try
    for i = 0 to Array.length tab - 1 do
      for j = 0 to Array.length tab.(i) - 1 do
        if est_min_local tab i j then raise (Trouve (i,j))
      done
    done; failwith "Ce cas n'arrive jamais sauf s'il n'y a pas d'élément"
  with Trouve (i,j) -> (i,j);;
```

La complexité est linéaire en la taille de la matrice, donc en la somme des tailles de ses lignes.

Question P2.7 :

La complexité est la même si on crée la matrice correspondant à la liste (temps linéaire en la taille totale) puis on appelle la fonction précédente (idem).

Cependant, en imposant de conserver la structure de liste, un calque de la fonction précédente nécessiterait d'écrire une fonction d'accès au i-ième élément de la j-ième liste, et un tel accès demande $i + j$ opérations pour remonter cet élément en tête, d'où une complexité pour une liste de n listes de taille m en $\mathcal{O}((m + n)mn)$.

Bien entendu, il existe une autre solution : rechercher le minimum global de la liste de listes, ce qui prend un temps linéaire en la taille totale.

Question P2.8 :

Cet exercice est sans nul doute le plus long et le plus difficile, et l'indication n'est pas de trop. Par conséquent, séparer en plusieurs fonctions est recommandé.

À ce titre, on réutilise la fonction `est_min_local`.

```
exception Min of int * int;;

let min_fenetre mat debl finl debc finc =
  let milieul = (debl+finl)/2 and milieuc = (debc+finc)/2 in
  let l = ref debl and c = ref debc in
    for j = debc to finc do
      if est_min_local mat debl j then raise (Min (debl, j));
      if est_min_local mat milieul j then raise (Min (milieul, j));
      if est_min_local mat finl j then raise (Min (finl, j));
      if mat.(debl).(j) < mat.(!l).(!c) then begin l := debl; c := j end;
      if mat.(milieul).(j) < mat.(!l).(!c) then begin l := milieul; c := j end;
      if mat.(finl).(j) < mat.(!l).(!c) then begin l := finl; c := j end
    done;
    for i = debl+1 to finl-1 do
      if est_min_local mat i debc then raise (Min (i, debc));
      if est_min_local mat i milieuc then raise (Min (i, milieuc));
      if est_min_local mat i finc then raise (Min (i, finc));
      if mat.(i).(debc) < mat.(!l).(!c) then begin l := i; c := debc end;
      if mat.(i).(milieuc) < mat.(!l).(!c) then begin l := i; c := milieuc end;
      if mat.(i).(finc) < mat.(!l).(!c) then begin l := i; c := finc end
    done;
    (!l, !c);;

let min_local_dicho mat =
  let n = Array.length mat in (* On suppose qu'on dispose d'une vraie matrice carrée. *)
    let rec aux debl finl debc finc =
      let milieul = (debl+finl)/2 and milieuc = (debc+finc)/2 in
        try
          let (ligne, colonne) = min_fenetre mat debl finl debc finc in
            if ligne = debl || (ligne = milieul && mat.(ligne).(colonne) > mat.(ligne-1).(colonne))
              then if colonne <= milieuc then aux debl milieul debc milieuc
              else aux debl milieul milieuc finc
            else if ligne = finl || ligne = milieul
              then if colonne <= milieuc then aux milieul finl debc milieuc
              else aux milieul finl milieuc finc
            else if colonne = debc (colonne = milieuc && mat.(ligne).(colonne) > mat.(ligne).(colonne-1))
              then if ligne <= milieul then aux debl milieul debc milieuc
              else aux milieul finl debc milieuc
            else
              if ligne <= milieul then aux debl milieul milieuc finc
              else aux milieul finl milieuc finc
        with Min (l, c) -> (l, c)
    in aux 0 (n-1) 0 (n-1);;
```

Ce code présente une particularité intéressante : la signature de la fonction auxiliaire n'est pas donnée par des cas de base vu qu'il n'y en a pas, c'est seulement le rattrapage d'exceptions qui permet de l'obtenir.
Il est nécessaire d'étudier toute la fenêtre et pas seulement la croix de séparation avec cet algorithme, car le minimum de cette croix peut être plus grand que le minimum de la fenêtre, et dans la zone qu'on étudierait alors, il est possible d'avoir une sorte de gradient dirigeant hors de la zone sans y trouver de minimum.

Question P2.9 :

Puisqu'on n'étudie qu'un des carrés et qu'on cherche aussi un minimum global sur une fenêtre avec $6n-9$ éléments, on a la formule $c_n = c_{\frac{n}{2}} + \mathcal{O}(n)$, donc d'après le Master Theorem le terme de droite est prépondérant et $c_n = \mathcal{O}(n)$.

Question P2.10 :

L'idée de couper en deux reste pertinente, et il est certes pratique pour le programmeur de découper toujours dans le même sens, jusqu'à ne plus avoir par exemple qu'une ligne, mais dans ce cas la complexité sera plus élevée.
La formule devient $c_m = c_{\frac{m}{2}} + \mathcal{O}(\sqrt{m})$ en alternant les séparations en deux horizontalement et verticalement afin que la taille de la fenêtre soit effectivement en $\mathcal{O}(\sqrt{m})$, soit $c_m = \mathcal{O}(\sqrt{m})$ (comme dans le cas précédent, donc), ou alors $c_l = c_{\frac{l}{2}} + \mathcal{O}(\sqrt{m})$ (avec $c_1 = \mathcal{O}(\sqrt{m})$) car le nombre d'opérations à chaque étape est linéaire en le nombre de lignes (par exemple) de la matrice de départ, et ici le Master Theorem ne peut pas vraiment s'appliquer mais une analyse similaire à l'écriture de sa preuve donne $c_m = \mathcal{O}(\sqrt{m}\log(m))$.

Question P2.11 :

La solution est en temps constant : on prend un élément arbitraire, disons un coin. Soit c'est un `false`, donc un minimum local, soit c'est un `true` dont les seuls voisins sont des `true`, et c'est encore un minimum local, soit c'est un `true` dont un voisin est un `false`, qui est, lui, un minimum local.

```
let min_local_bool mat =
  if not mat.(0).(0) then (0, 0) (* si mat ou mat.(0) est vide, tant pis, il y aura une erreur)
  else if Array.length mat > 1 && not mat.(1).(0) then (1, 0)
  else if Array.length mat.(0) > 1 && not mat.(0).(1) then (0, 1)
  else (0, 0);;
```

Question P2.12 :

L'algorithme est simple : on part d'une position quelconque et on suit un chemin dans la matrice en prenant comme successeur le voisin le plus petit, en donnant priorité à la position actuelle. Ce chemin se stabilise sur un minimum local, et le nombre d'étapes est au plus la valeur du premier élément visité, donc au plus le maximum de la matrice, d'où la complexité. En effet, puisque la position actuelle est prioritaire, un déplacement garantit une décroissance stricte, et l'algorithme s'arrête dès que l'on reste sur place.

```
let plus_petit_voisin tab i j =
  let n = Array.length tab and l = ref i and c = ref j in
      if i > 0 && j < Array.length tab.(i-1) && tab.(i-1).(j) < tab.(!l).(!c)
        then begin l := i-1; c := j end;
      if i < n-1 && j < Array.length tab.(i+1) && tab.(i+1).(j) < tab.(!l).(!c)
        then begin l := i+1; c := j end;
      if j > 0 && tab.(i).(j-1) < tab.(!l).(!c)
        then begin l := i; c := j-1 end;
      if j < Array.length tab.(i)-1 && tab.(i).(j+1) < tab.(!l).(!c)
        then begin l := i; c := j+1 end; (!l, !c);;

let min_local_borne mat =
  let rec parcours l c = let (ll, cc) = plus_petit_voisin mat l c in
      if l = ll && c = cc then (l, c) else parcours ll cc
  in parcours 0 0;;
```