I Utilisation des tableaux

I.1 Fonctions avancées

Exercice 1

```
let map f t =
   let n = Array.length t in
   let t' = Array.make n (f t.(0)) in
   for i = 1 to n - 1 do
      t'.(i) <- f t.(i)
   done;
   t';;

let iteri f t =
   for i = 0 to Array.length t - 1 do
      f i t.(i)
   done;;

let fold_left f a t =
   let x = ref a in
   for i = 0 to Array.length t - 1 do
      x := f !x t.(i)
   done;
   !x;;
```

Pour la fonction `map`, on recrée un tableau qu'on remplit avec les images. Pour `fold_left`, on utilise une référence pour garder en mémoire la valeur itérée.

I.2 Matrices

Exercice 2

On écrit une fonction qui échange deux valeurs dans la matrice et on utilise cette fonction plusieurs fois. Cela évite de créer une nouvelle matrice. On fait attention aux indices pour éviter de faire des échanges plusieurs fois.

```
let echange m i j =
   let x = m.(i).(j) in
   m.(i).(j) <- m.(j).(i);
   m.(j).(i) <- x;;

let transpose m =
   let n = Array.length m in
   for i = 0 to n - 2 do
      for j = i + 1 to n - 1 do
         echange m i j
      done
   done;;
```

Exercice 3

On commence par créer une matrice de la bonne taille, puis on fait le calcul mathématique.

```
let mult a b =
   let m = Array.length a and n = Array.length b and p = Array.length b.(0) in
   let c = Array.make_matrix m p 0 in
   for i = 0 to m - 1 do
      for j = 0 to p - 1 do
         for k = 0 to n - 1 do
            c.(i).(j) <- c.(i).(j) + a.(i).(k) * b.(k).(j)
         done
      done
   done;
   c;;
```

I.3 Files d'attente

Exercice 4

1. On pense bien à rendre les indices mutables.

```
type 'a file = {mutable p : int; mutable d : int; tableau : 'a array};;
```

2. Les indices sont initialisés à 0.

```
let creer n a = {p = 0; d = 0; tableau = Array.make n a};;
```

3. On applique les tests décrits :

```
let vide f = f.d = f.p;;

let pleine f = f.p = (f.d + 1) mod (Array.length f.tableau);;
```

4. On pense à faire les modifications d'indices modulo n.

```
let enfile a f =
   if pleine f then failwith "File pleine"
   else begin
      f.tableau.(f.d) <- a;
      f.d <- (f.d + 1) mod (Array.length f.tableau)
   end;;

let defile f =
   if vide f then failwith "File vide"
   else begin
      let a = f.tableau.(f.p) in
      f.p <- (f.p + 1) mod (Array.length f.tableau);
      a
   end;;
```

I.4 Dictionnaires

Exercice 5

Cela correspond aux fonctions écrites en TP, avec un type différents (car mutable). Comme on travaille avec des listes, il est pertinent d'utiliser des fonctions auxiliaires récursives.

```
let creer () = {liste = []};;

let ajouter (a, b) d =
   d.liste <- (a, b) :: d.liste;;

let supprimer a d =
   let rec supp = function
      | [] -> []
      | (x, _) :: q when x = a -> q
      | (x, y) :: q -> (x, y) :: (supp q) in
   d.liste <- supp d.liste;;

let modifier (a, b) d =
   let rec modi = function
      | [] -> []
      | (x, _) :: q when x = a -> (a, b) :: q
      | (x, y) :: q -> (x, y) :: (modi q) in
   d.liste <- modi d.liste;;

let lire a d =
   let rec lir = function
      | [] -> failwith "Non trouvé"
      | (x, y) :: q when x = a -> y
      | _ :: q -> lir q in
   lir d.liste;;
```

II Analyse d'algorithme

II.1 Terminaison

Exercice 6

On pose $v_n = 2q_n + 3c_n$ où n est le numéro du passage dans la boucle. On vérifie :

- $v_0 = 2p$. Or, on ne passe dans la boucle que si $q_0 > 0$, c'est-à-dire si $v_0 > 0$.
- Pour $n \in \mathbb{N}^*$, $v_n = 2q_n + 3c_n$. Or, $c_n \geq 0$ et on ne continue dans la boucle que si $q_n > 0$ (car sinon, la boucle termine).
- Pour $n \in \mathbb{N}$, $v_{n+1} = 2q_{n+1} + 3c_{n+1}$. On distingue alors :
  - Si $c_n = 0$, on a $q_{n+1} = q_n - 2$ et $c_{n+1} = 1$, soit $v_{n+1} = 2q_n + 3c_n - 1 = v_n - 1$.
  - Si $c_n = 1$, on a $q_{n+1} = q_n + 1$ et $c_{n+1} = 0$, soit $v_{n+1} = 2q_n + 3c_n - 1 = v_n - 1$.

Il s'agit donc bien d'un variant de boucle.

II.2 Correction

Exercice 7

Il est claire que la fonction `echange` échange de place deux valeurs dans le tableau. On vérifie les propriétés de l'invariant de boucle :

- Initialisation : Si $j = 0$, alors $i = 0$, et les deux ensembles de la propriété sont vides. La propriété est bien vérifiée.
- Conservation : Supposons le résultat vrai pour $j - 1 \in \mathbb{N}$ fixé. Deux cas se présentent alors après le passage dans la boucle pour j :
  * $t[j] \geq p$, auquel cas on ne fait rien. La première des deux propriétés est toujours vérifiée (on n'a pas modifié l'ensemble) et la seconde est également toujours vérifiée (car $t[j] \geq p$).
  * $t[j] < p$, auquel cas on augmente la valeur de i, puis on échange les éléments de positions i et j. Ainsi, dans le premier ensemble, on a rajouté un seul élément, qui est $t[j]$ et se retrouve à la position de l'ancienne valeur de i plus un. La propriété est toujours vrai. Dans le second ensemble, qui est toujours de même taille, on a juste mis en position j l'élément qui était en ancienne position $i + 1$, qui était bien supérieur à p. La propriété est vérifiée.
- Terminaison : Après le dernier passage dans la boucle, la propriété est vraie pour tous les éléments de $[\![1; n-1]\!]$. On échange alors les positions 0 et i, et comme l'élément en position i est inférieur à p, on en conclut que l'algorithme renvoie bien le résultat attendu.

II.3 Complexité

Exercice 8

On teste successivement les valeurs 0, 1, -1, 2, -2, ... jusqu'à trouver la valeur k. Le nombre de tests effectués sera de l'ordre de $2k$ (éventuellement plus un, pour 0 et les négatifs), donc bien linéaire.

```
let trouve_k f =
   let x = ref 0 in
   while f !x = 0 do
      if !x > 0 then
         x := - !x
      else
         x := - !x + 1
   done;
   !x;;
```

Exercice 9

1. Si $k = 1$, la stratégie optimale consiste à lâcher l'œuf du premier étage, et tant qu'il ne s'est pas cassé, à augmenter l'étage de 1 et recommencer. La complexité est de n lâchers dans le pire des cas.
   Si $k = n$, on a autant d'œufs que l'on veut. On peut ici faire une recherche dichotomique, et obtenir une complexité en au plus $\lceil \log(n+1) \rceil$ lâchers.
2. On sépare les étages en $\lceil \sqrt{n} \rceil$ paquets de $\lceil \sqrt{n} \rceil$ étages. Le premier œuf sert à déterminer dans quel paquet se trouve l'étage critique (on commence au sommet du premier paquet, et on augmente d'un paquet à chaque lâcher réussi), et le deuxième œuf à déterminer de la même manière l'étage critique dans le paquet. Cela nécessite au pire $2\lceil \sqrt{n} \rceil$ lâchers.

III Programmation dynamique

III.1 Plus longue sous-séquence croissante

III.2 Multiplication chaînée de matrices

Exercice 10

1. Il faut calculer chacun des $m \times p$ éléments de la matrice produit. Le calcul de chaque élément se fait en n multiplications, soit un total de mnp.
2. Pour $A \times ((B \times C) \times D)$, on effectue $20 \times 1 \times 10 + 20 \times 10 \times 100 + 50 \times 20 \times 100 = 200 + 20000 + 100000 = 120200$ multiplications. Pour $(A \times B) \times (C \times D)$, on effectue $50 \times 20 \times 1 + 1 \times 10 \times 100 + 50 \times 1 \times 100 = 1000 + 1000 + 5000 = 7000$ multiplications.
3. $A_i \times A_{i+1} \ldots \times A_j = A_i \times \ldots \times A_k \times A_{k+1} \times \ldots \times A_j$, pour tout $k \in [\![i; j-1]\!]$. On en déduit que $C(i,j) = \min\{C(i,k) + C(k+1,j) + m_{i-1} \times m_k \times m_j | k \in [\![i; l+1]\!]\}$.
4. On applique ce qui est décrit ci-dessus, en faisant attention aux dimensions (le tableau de dimensions est de taille un de plus que le nombre de matrices).

```
let chaine m =
   let n = Array.length m - 1 in
   let c = Array.make_matrix (n + 1) (n + 1) 0 in
   for ecart = 1 to n - 1 do
      for i = 1 to n - ecart do
         let j = i + ecart in
         let x = m.(i - 1) * m.(j) in (* Pour éviter de refaire le calcul *)
         c.(i).(j) <- c.(i).(i) + c.(i + 1).(j) + x * m.(i);
         for k = i + 1 to j - 1 do
            let y = c.(i).(k) + c.(k + 1).(j) + x * m.(k) in
            c.(i).(j) <- min c.(i).(j) y
         done
      done
   done;
   c.(1).(n);;
```

5. On a 3 boucles `for` imbriquées. Sans rentrer dans les détails de calculs (car les boucles ne sont pas toujours de la même taille), on arrive facilement à se convaincre que la complexité est $O(n^3)$.
6. Pour déterminer le parenthésage, on peut renvoyer un tableau (ou une liste) contenant les indices des dimensions où on effectue la multiplication. Dans l'exemple précédent, le produit $(A \times B) \times (C \times D)$ est décrit par le tableau `[|1;3;2|]`. Pour ce faire, on peut créer une matrice `p` tel que `p.(i).(j)` contient l'indice `k` indiquant le découpage à faire pour calculer $A_i \times A_{i+1} \ldots \times A_j$. On peut alors remonter le calcul une fois `p.(1).(n)` déterminé.

III.3 Problème du sac à dos (Knapsack)

Exercice 11

Pour calculer $K(w,j)$, il suffit d'envisager les cas où l'objet j est choisi et où il ne l'est pas. On a :

$$K(w,j) = \max(K(w - w_j, j-1), K(w, j-1))$$

On en déduit la fonction suivante, où `w` désigne le tableau de poids, `v` désigne le tableau de valeurs et `wmax` le poids maximal.

```
let knapsack w v wmax =
   let n = Array.length w in
   let k = Array.make_matrix (wmax + 1) (n + 1) 0 in
   for j = 0 to n - 1 do
      for p = 1 to wmax do
         if w.(j) > p then
            k.(p).(j + 1) <- k.(p).(j)
         else
            k.(p).(j + 1) <- max k.(p).(j) (k.(p - w.(j)).(j) + v.(j))
      done
   done;
   k.(wmax).(n);;
```

III.4 Distance d'édition

Exercice 12

Il suffit d'envisager les cas d'une insertion, suppression ou substitution. On a :

$$E(i,j) = \min\{1 + E(i-1,j), 1 + E(i,j-1), 1 - \delta_{u_i,v_j} + E(i-1,j-1)\}$$

où $\delta_{k,\ell}$ désigne le symbole de Kronecker (qui vaut 0 ou 1 selon que k et ℓ soient différents ou non). On en déduit la fonction :

```
let levenshtein u v =
   let m = String.length u and n = String.length v in
   let e = Array.make_matrix (m + 1) (n + 1) 0 in
   for i = 0 to m do
      e.(i).(0) <- i
   done;
   for j = 1 to n do
      e.(0).(j) <- j
   done;
   for i = 1 to m do
      for j = 1 to n do
         let delta = if u.[i - 1] = v.[j - 1] then 1 else 0 in
         let x = e.(i - 1).(j) in
         let y = e.(i).(j - 1) in
         let z = e.(i - 1).(j - 1) in
         e.(i).(j) <- 1 + min (min x y) (- delta + z)
      done
   done;
   e.(m).(n);;
```