DM - L'algorithme de Kaprekar

Question 1 *Écrire une fonction* `log : int -> int -> int` *telle que* `log b n` *renvoie le plus entier k tel que* $n < b^k$ *(avec* $b \geq 2$*).*

On peut distinguer de manière de procéder : de façon croissante (on calcule les puissances de la base jusqu'à dépasser n) ou de façon décroissante (on divise n par la base jusqu'à aboutir à 0).

Voici la version croissante de façon itérative :

```
let rec log b n =
  let k = ref 0
  and p = ref 1 in
  while !p <= n do
    incr k;
    p := b * !p
  done;
  !k
;;
```

Une version récursive de cela passe utilise une fonction auxilliaire :

```
let log2 b n =
  let rec aux p =
    if p > n then 0 else 1 + aux (p * b)
  in
  aux 1
;;
```

que l'on peut facilement rendre récursive terminale :

```
let log2bis b n =
  let rec aux p k =
    if p > n then k else aux (p * b) (k + 1)
  in
  aux 1 0
;;
```

Une version décroissante consiste à trouver le plus petit entier k tel que $\frac{n}{b^k} < 1$ soit, puisque l'on travaille avec des entiers, tel que l'on obtienne 0 :

```
let log3 b n =
  let v = ref n
  and k = ref 0 in
```

```
  while !v > 0 do
    v := !v / b;
    incr k
  done;
  !k
;;
```

La version récursive est la suivante :

```
let rec log4 b n = if n = 0 then 0 else 1 + log4 b (n / b) ;;
```

Question 2 *Écrire une fonction* `vers_tableau : int -> int -> int array` *telle que* `vers_tableau b n` *renvoie le tableau contenant la décomposition de* `n` *en base* $b \geq 2$.

À chaque étape, on calcule le reste et le quotient de la division euclidienne par `b`.

```
let vers_tableau b n =
  let taille = log b n in
  let tab = Array.make taille 0 in
  let v = ref n in
  for i = 0 to taille - 1 do
    tab.(i) <- !v mod b;
    v := !v / b
  done;
  tab
;;
```

On évitera à tout prix de passer par les flottants (ce n'est vraiment pas l'esprit, on fait ici des calculs exactes), et une formule du type `(n / (pow b k)) mod b` voire `(n mod (pow b (k + 1))) / (pow b k)` est délicate car il faut alors calculer efficacement toutes les puissances de `b`.

Question 3 *Écrire de même la fonction*

`depuis_tableau : int -> int array -> int`.

Une façon de procéder est de stocker les puissances successives de la base :

```
let depuis_tableau b tab =
  let n = ref 0
  and p = ref 1 in
  for i = 0 to Array.length tab - 1 do
    n := !n + (tab.(i) * !p);
    p := !p * b
  done;
  !n
```

```
;;
```

Notons qu'il est très inefficace de recalculer les puissances de la base en « recommançant de 0. »

On utilise l'algorithme de Horner pour une efficacité optimale, qui repose sur l'écriture :

$$a+10b+10^2c+10^3d+\cdots = a+10\Big(b+10\big(c+10(d+\cdots)\big)\Big)$$

```
let depuis_tableau b tab =
  let v = ref 0 in
  for i = Array.length tab - 1 downto 0 do
    v := (b * !v) + tab.(i)
  done;
  !v
;;
```

Notons l'utilisation de **`downto`** pour faire une boucle **`for`** décroissante.

Question 4 *Écrire la fonction*

```
kaprekar : int -> int
```

qui effectue cette opération.

On verra attention à la gestion du ou des tableaux. On peut n'avoir qu'un tableau que l'on trie dans un sens, on récupère la valeur correspondante, puis on le trie dans l'autre sens et on récupère la valeur correspondante.

```
let kaprekar n =
  let t = vers_tableau 10 n in
  Array.sort (fun x y -> y - x) t;
  let a = depuis_tableau 10 t in
  Array.sort (fun x y -> x - y) t;
  let b = depuis_tableau 10 t in
  b - a
;;
```

Question 5 *Écrire une fonction* `point_fixe : ('a -> 'a) -> 'a -> 'a`.

Il est tentant d'écrire quelque chose comme

```
let rec point_fixe f x = if x = f x then x else point_fixe f (f x) ;;
```

mais on calcule deux fois `f x` (ce qui peut être coûteux) alors que c'est inutile. On aura donc intérêt à écrire :

```
let rec point_fixe f x =
  let y = f x in
  if x = y then x else point_fixe f y
;;
```

Il est bien sûr d'en faire une version itérative (mais c'est beaucoup plus long à écrire) :

```
let point_fixe f x =
  let x = ref x
  and y = ref (f x) in
  while !x <> !y do
    x := !y;
    y := f !x
  done;
  !x
;;
```

Question 6 *En déduire une fonction* `point_fixe_kaprekar : int -> int` *qui renvoie le point fixe obtenu à partir de l'entier donné en appliquant la transformation de Kaprekar.*

L'idée est bien sûr d'utiliser la fonction de la question précédente. Il suffit d'écrire

```
let point_fixe_kaprekar i = point_fixe kaprekar i ;;
```

Notons que l'on peut aussi écrire simplement

```
let point_fixe_kaprekar = point_fixe kaprekar ;;
```

Question 7 *Calculer le nombre d'entiers* $n \in [\![1000, 9999]\!]$ *dont le point fixe précédent est égal à* 6174.

On peut écrire une petite fonction qui fait cela.

```
let compte () =
  let n = ref 0 in
  for i = 1000 to 9999 do
    if point_fixe kaprekar i = 6174 then incr n
  done;
  !n
;;
```

On a alors

```
# compte () ;;
- : int = 8923
```