Algorithmes sur les tableaux

I. Différents types de parcours

On rappelle que la longueur d'un tableau s'obtient à l'aide de la fonction

```
Array.length : 'a array -> int.
```

1. Parcours complet

Si l'on sait que l'on va devoir parcourir toutes les cases d'un tableau, une boucle **`for`** s'impose.

Exercice 1

1. Écrire une fonction

   ```
   minimum : int array -> int
   ```

   qui renvoie le minimum du tableau non vide d'entiers passé en argument.

2. Prouver la correction de la fonction en explicitant un invariant de boucle.

Exercice 2 Faire de même pour des fonctions :

- ⋆ `somme : int array -> int` qui calcule la somme des éléments d'un tableau d'entiers ;
- ⋆ `occurences : 'a array -> 'a -> int` qui, étant donné un tableau `t` et une valeur `v`, calcule le nombre d'occurences de `v` dans `t`.

2. Parcours partiels

Si l'on ne va pas parcourir un tableau dans son intégralité, mieux vaut utiliser une boucle **`while`**.

Exercice 3

1. Écrire une fonction :

   ```
   pour_tout : ('a -> bool) -> 'a array -> bool
   ```

   telle que `pour_tout p t` renvoie `true` si tous les éléments de `t` vérifient le prédicat `p`, et `false` sinon.

2. Prouver sa correction.

3. Parcours simultané de deux tableaux ou plus

Exercice 4 Écrire une fonction

```
en_commun_2 : int array -> int array -> int
```

qui, étant donné deux tableaux d'entiers strictement croissants `t1` et `t2`, indique le nombre d'éléments en communs.

Exercice 5 Écrire une fonction

```
est_somme : int array -> int array -> int -> bool
```

qui, étant donné deux tableaux d'entiers croissants `t1` et `t2` et un entier `n` indique si cet entier peut être écrit comme somme d'un élément de `t1` et d'un élément de `t2`.

On essaiera d'avoir la meilleure complexité possible.

On veut avoir, par exemple :

```
# est_somme [| 1; 3; 4; 8|] [| 2; 5; 6; 9|] 12 ;;
- : bool = true
# est_somme [| 1; 3; 4; 8|] [| 2; 5; 6; 9|] 11 ;;
- : bool = false
```

4. Parcours dichotomique

Exercice 6 Écrire une fonction

```
occurences_croissant : 'a array -> 'a -> int
```

qui, étant donné un tableau d'éléments croissants et une valeur, renvoie le nombre d'occurences de cette valeur dans le tableau.

On essaiera d'avoir une complexité logarithmique en la longueur du tableau dans tous les cas.

II. Le jeu 2048

Vous connaissez sans doutes le jeu 2048 (sinon, allez rapidement le découvrir). Nous allons essayer de programmer le mode de déplacement des cases. Il comporte deux éléments principaux. Pour un mouvement vers la gauche, par exemple,

⋆ toutes les cases non vides sont décalées au maximum vers la gauche ;

⋆ deux cases consécutives avec la même valeur

Par exemple, en représentant les cases d'une ligne par un tableau d'entier (les cases vides correspondant aux valeurs `0`), en partant du tableau

```
[| 1; 0; 2; 2|]
```

on obtient à l'issue vers la gauche :

```
[| 1; 4 ; 0; 0|]
```

Nous allons commencer par le décalage vers la gauche :

Question 7 Écrire une fonction

```
decaler_gauche : int array -> unit
```

qui, étant donné un tableau `t`, « décale » toutes les valeurs non nulles vers la gauche. On veut par exemple :

```
# let t = [| 1; 2; 0; 1; 0; 0; 1; 0; 4|] ;;
val t : int array = [| 1; 2; 0; 1; 0; 0; 1; 0; 4|]
# decaler_gauche t ;;
- : unit = ()
# t ;;
- : int array = [|1; 2; 1; 1; 4; 0; 0; 0; 0|]
```

Passons maintenant à l'étape de fusion.

Question 8 Écrire une fonction

```
fusionner_gauche : int array -> unit
```

qui parcours le tableau de gauche à droite, tout couple de cases adjacentes contenant la même valeur, double la valeur de la case de gauche et mets l'autre case à zéro. Par exemple, on doit avoir :

```
# let t = [| 1; 2; 2; 2; 4; 4; 1|] ;;
val t : int array = [|1; 2; 2; 2; 4; 4; 1|]
# fusionner_gauche t ;;
- : unit = ()
# t ;;
- : int array = [|1; 4; 0; 2; 8; 0; 1|]
```

Question 9 En déduire une fonction `a_gauche : int array -> unit` pour effectuer le mouvement complet.

Question 10 Finir de programmer le jeu. Vous pouvez aller voir le code source

original, en Javascript mais assez lisible.