# TD2 : Algdem

**Exercice 1 : Tri fusion**

Voici une implémentation sur les listes `Haskell` de l'algorithme de tri par fusion.

```
separe :: [Int] -> [Int] -> ([Int], [Int])
separe l []            = ([],l)
separe l [_]           = ([],l)
separe [] _            = ([],[])
separe (x:l) (_:_:cpt) = let (l1,l2) = (separe l cpt) in (x:l1,l2)

fusionne :: [Int] -> [Int] -> [Int]
fusionne [] l = l
fusionne l [] = l
fusionne (x1:l1) (x2:l2) | x1 <= x2  = x1 : fusionne l1 (x2:l2)
                         | otherwise = x2 : fusionne (x1:l1) l2

triFusion :: [Int] -> [Int]
triFusion []  = []
triFusion [x] = [x]
triFusion l   = let (l1,l2) = separe l l in fusionne (triFusion l1) (triFusion l2)
```

L'algorithme est présenté à l'aide de plusieurs fonctions :

— `separe l1 l2` découpe la liste `l1` en deux listes `(l11, l12)` telle que la longueur de `l11` est la moitié de celle de `l2`.

— `fusionne` qui fusionne deux listes triées en une liste triée contenant les mêmes éléments que les deux listes passées en argument.

— `triFusion` qui effectue le tri en découpant la liste passée en argument en deux, en triant récursivement les deux listes obtenues et en fusionnant ces listes triées.

Nous allons étudier les fonctions qui concourent à cette implémentation du tri fusion.

**Q 1.1 Couper une liste en deux**

Nous allons commencer par étudier la fonction `separe`.

**Q 1.1.1 Exécuter `separe`.** Évaluez pas à pas l'expression `separe [1,3,2,7,1] [5,3,4,1]`

separe [1,3,2,7,1] [5,3,4,1] =

let (l1,l2) = (separe [3,2,7,1], [4,1]) in (1:l1, l2)

let (l1,l2) = (let (l1,l2) = (separe [2,7,1] [ ] in (1:l1,l2)) in (3:l1,l2)

= let (l1,l2) = let (l1,l2) = ((), [2,7,1]) in (1:l1,l2)| in (3:l1,l2)

= ([1], [2,7,1]) in (3:l1,l2)

= ([1,3], [2,7,1]).

**Q 1.1.3 Spécification et démonstration de `separe`.** Montrez par induction sur `length l1` que, en posant `(l11, l12) = separe l1 l2`, nous avons :

1. `l11 ++ l12 = l1`
2. `length l11 = min (length l1) (length l2 `quot` 2)`

Voici les définitions de `++`, `length` et `min`.

```
length :: [a] -> Int
length [] = 0
length (_:l) = 1 + length l

(++) :: [a] -> [a] -> [a]
[] ++ l = l
(x:l1) ++ l2 = x:(l1 ++ l2)

min :: Int -> Int -> Int
min a b | a <= b   = a
        |otherwise = b
```

Le variant sera length l1, et en posant

(l11, l12) = separe l1 l2, on a pour HI

HI : ∀ l1'. length l1' < lenght l1 =>

l1' = (l11' ++ l12')

lenght l11' = min (lenght l1') (lenght l2' `quot` 2)

Cas 1 : l2 = []

separe l1 l2 = ([], l2) = (l1, l2)

On a bien l1 = [] ++ l2 = [] ++ l1

On a lenght l1 = lenght [] = min (lenght l1) (lenght l2 `quot` 2)

≤ 0 ok

Cas 2 et 3 analogues ...

Cas 4 : lenght l1 > 0 et lenght l2 ≥ 2

separe (x : l1) (_ : _ : l2) = let (l1, l2) = separe (l1, l2) in (x : l1, l2)

= (x : l1, l2)

= (x : (separe (l1, l2))[0], (separe (l1, l2))[-1])

Comme lenght (x : l1) > 0, on a lenght l1 ≥ 0

et lenght l1 < lenght (x : l1), on applique HI.

- l1 = l11 ++ l12
- lenght l11 = min (lenght l1) ((lenght l2 `quot` 2)

On "Remonte" le cas :

- xs :: l1 = x :: l11 ++ l12 (par définition + vu en haut)
- lenght xs :: l1 = min (lenght xs :: l1) (lenght l2 `quot` 2)

CQFR.