# TD3 : AlgDem

```
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
```

**Q 1.2 Fusionner des listes triées**
Nous allons maintenant étudier la fonction `fusionne`.

**Q 1.2.1 Exécuter `fusionne`.** Exécutez pas à pas le calcul de `fusionne [1,3] [1,2,7]`.

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

**Q 1.2.3 Spécification et démonstration de `fusionne`.** Nous allons maintenant étudier la fonction `fusionne`.
Pour le calcul de `fusionne l1 l2`, nous utilisons la spécification suivante :
**Pré-condition** `l1` et `l2` sont ordonnées par ordre croissant (`triée(l1)` et `triée(l2)`)
**Post-condition** `triée(fusionne l1 l2)` et `perm(fusionne l1 l2, l1 ++ l2)`.
Montrez par induction sur `length l1 + length l2` que `fusionne l1 l2` respecte bien sa spécification.

H1 : ∀ (l1', l2', length l1' + length l2' < length l1 + length l2
∧ triée (l1') ∧ triée (l2') =>
(P1) (P2)
- triée (fusionne l1' l2)
- perm (fusionne l1' l2, l1' ++ l2)

} (H1)

Cas 1 : l1 = [3]
On a : fusionne l1 l2 = l2 (E)
Comme on a triée (l2), il vient que triée (fusionne l1 l2)
et d'après (E), on a aussi perm (fusionne l1 l2, l1 ++ l2).

Cas 2 analogue.

Cas 3 : l1 = x1 :: l1' (C31) ∧ l2 = x2 :: l2' (C32)

Cas 3.1 : x1 ≤ x2 (H1)

Sous ces conditions, nous avons :

fusionne l1 l2 = x1 :: fusionne l1' l2 (par H1)

Comme length l1' + length l2 < length l1 + length l2, on a que le variant décroit.

Comme on a triée (l1) par (P1) on a nécessairement triée (l1'), car retirer le premier élément d'une liste triée garde la liste triée. On a aussi triée (l2) par (P2)

On peut donc appliquer (HI)

- triée ( fusionne l1' l2)
- perm (fusionne l1' l2 , l1' ++ l2)

Or, comme l1 = x1 :: l1', x1 est plus petit que tous les éléments de l1' (car triée (l1)).

x1 est également plus petit que tous les éléments de l2 (par (H1))

Donc il vient que x1 est le plus petit des éléments de fusionne l1' l2.

Il vient que :

triée (fusionne l1' l2) <=> triée (fusionne l1 l2)

perm facile.

Cas 3.2 analogue

Preuve finie.

**Q 1.3 Le tri par fusion**

Nous sommes maintenant en position de démontrer que la fonction `triFusion` réalise bien le tri des listes. Montrez par induction sur `length l` que `triée(triFusion l)` et `perm(triFusion l, l)`.

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

∀l'. length l' < length l =>

- triée (triFusion l')
- perm (triFusion l', l')

} (HI)

Cas 1 et 2 évident (en supposant triée [Σ])

Cas 3 : length l > 1 (C3)

En posant (l1, l2) = separe l l

On a que :

length l1 < length l et

length l2 < length l .

On a alors :

fusionne (triFusion l1) (triFusion l2)

En appliquant (HI), il vient que :

- triée (triFusion l1)
- triée (triFusion l2)
- perm (triFusion l1, l1)
- perm (triFusion l2, l2)

Par la question 1.2.3, il vient que

- tri (fusionne (triFusion l1) (triFusion l2))
- perm (fusionne (triFusion l1) (triFusion l2),
  (triFusion l1)++(triFusion l2)), ce qui conclut la preuve.

Dans cet exercice nous allons travailler avec une représentation des nombres par la liste de leurs facteurs premiers. Par exemple, nous représenteront le nombre `36` par la liste triée de ses facteurs premiers (avec leur multiplicité) `[2,2,3,3]`.

Nous allons ainsi étudier les fonctions suivantes :

```
produit :: [Int] -> Int
produit []     = 1
produit (x:l) = x * produit l

pgcd :: [Int] -> [Int] -> [Int]
pgcd [] _ = []
pgcd _ [] = []
pgcd (x1:l1) (x2:l2) | x1 == x2  = x1 : pgcd l1 l2
                     | x1 < x2   = pgcd l1 (x2:l2)
                     | otherwise = pgcd (x1:l1) l2
```

**Q 2.1** Nous allons étudier la fonction `pgcd l1 l2` :

**Pré-condition** Les listes `l1` et `l2` sont triées (`triée(l1)` et `triée(l2)`) et ne contiennent que des nombres premiers positifs,

**Post-condition** `produit(pgcd l1 l2)` est le plus grand diviseur commun de `produit l1` et `produit l2`.

Montrez par induction que `pgcd` vérifie sa spécification, vous donnerez un variant pour faire cette démonstration.

Variant : length l1 + length l2

(HI) : $\forall$ l1', l2', length l1' + length l2' < length l1 + length l2

=> produit (pgcd l1 l2) est le pgcd de produit l1 et produit l2.

Cas 1 : l1 = []

pgcd l1 l2 = pgcd [] l2 = []

On a que $\forall b \in \mathbb{N}$, pgcd(1, b) = 1, donc ok.

Cas 2 pareil

Cas 3 : l1 = x1 :: l1' et l2 : x2 :: l2'.

Cas 31 : x1 == x2 (C31)

pgcd (x1 :: l1') (x2 :: l2') = x1 :: pgcd l1' l2'

On a length l1' + length l2' < length l1 + length l2

Donc on applique (HI)

produit (pgcd l1' l2') est le plus grand diviseur commun de produit l1' et produit l2'.

Disjonction de cas sur si $x1$ est le pgcd de $l1$ et $l2$.