TP 2 - Décomposition en facteurs premiers des entiers naturels

On pourra :

1. se reporter à la présentation du TP #2 faite lors du CM #4.
2. consulter les premières sections de :
   `https://fr.wikipedia.org/wiki/D%C3%A9composition_en_produit_de_facteurs_premiers`

Rappel : chaque décomposition est représentée par un liste de couples `(facteur, exposant)` et on définit alors les types synonymes suivants :

```
type Facteur = Int
type Exposant = Int
type Couple = (Facteur, Exposant)
type Decomposition = [Couple]
```

1. Proposer deux définitions équivalentes de la fonction `(rep2int xs)` qui retourne l'entier associé à une décomposition.

   ```
   > rep2int []   ==> 1
   > rep2int [(2,3), (5,2)]   ==> 200
   > rep2int [(2,1), (5,1), (7,3)]   ==> 3430
   ```

   (a) la première en utilisant une ZF-expression (listes en compréhension)

   (b) la seconde en utilisant une écriture récursive et le pattern-matching
   Indication :

   ```
   rep2intBis [] =
   rep2intBis ((x,n):xs) =
   ```

2. Définir la fonction `(estPremier xs)` qui détermine si une décomposition représente un nombre premier (sans utiliser `rep2int`).

   ```
   > estPremier [(2,3), (5,2)]   ==> False
   > estPremier [(2,1)]   ==> True
   ```

3. Proposer deux définitions équivalentes de la fonction `(pgcd xs ys)` qui calcule le pgcd (Plus Grand Commun Diviseur) de 2 entiers représentés par leur décomposition (sans utiliser `rep2int`).

   ```
   > pgcd [(2,3), (5,2)] [(2,1), (5,1), (7,3)]   ==> [(2,1), (5,1)]
   > pgcd  [(2,3), (5,2)] [(2,2)]   ==> [(2,2)]
   > pgcd [(2,3), (5,2)] [(7,3)]    ==> []
   ```

   (a) la première en utilisant une ZF-expression

   (b) la seconde en utilisant une écriture récursive et le pattern-matching
   Indications

   ```
   pgcdBis [] _ =
   pgcdBis _ [] =
   pgcdBis ((k1,d1):p1) ((k2,d2):p2)
      | k1 == k2 =
      |          =
      |          =
   ```

4. Définir la fonction `(ppcm xs ys)` qui calcule le ppcm (Plus Petit Commun Multiple) de 2 entiers représentés par leur décomposition (sans utiliser `rep2int`).

```
> ppcm [(2,3), (5,2)] [(7,3)]   ==> [(2,3),(5,2),(7,3)]
> ppcm [(2,3), (5,2)] [(2,2)]   ==> [(2,3),(5,2)]
> ppcm [(2,3), (5,2)] [(2,1), (5,1), (7,3)]   ==> [(2,3),(5,2),(7,3)]
```

Indication : on pourra s'inspirer de la définition de la fonction `(pgcdBis xs ys)`

5. Définir la fonction `(nbDiviseurs xs)` qui détermine le nombre de diviseurs du nombre entier dont la décomposition est `xs` (sans utiliser `rep2int`).

```
> nbDiviseurs [(2,1)]   ==> 2
> nbDiviseurs [(3,2)]   ==> 3
> nbDiviseurs [(2,1), (3,2)]  ==> 6
> nbDiviseurs [(2,1), (3,2), (7,3)]  ==> 24
```

Indications :

- *Quel est le nombre de diviseurs du nombre a^n lorsque a est un nombre premier ?*
- *Si 2 nombres a et b sont premiers entre eux, quel est le lien entre le nombre de diviseurs de $(a \times b)$ et les nombres de diviseurs des nombres a et b ?*
- *En déduire la définition de la fonction* ***`nbDiviseurs`****.*

6. On souhaite retrouver la liste des diviseurs d'un entier à partir de sa décomposition.

```
> diviseurs [(5,2)]   ==> [1,5,25]
> diviseurs [(2,3), (5,2)]    ==> [1,5,25,2,10,50,4,20,100,8,40,200]
> op (2,3) [1,5,25]   ==> [1,5,25, 2,10,50, 4,20,100, 8,40,200]
```

   (a) Que fait la fonction `(op n ys)` définie ci-dessous ?

```
op :: Couple -> [Int] -> [Int]
op (x,n) ys = [(x^i) * y | i <- [0..n], y <- ys]
```

   (b) En déduire la définition de `(diviseurs xs)`

7. **Premier exemple de traitement d'une liste infinie.**

   On définit la fonction suivante :

```
primes :: [Int]
primes =  sieve [2 .. ]
       where sieve (p:xs) = p : (sieve [x | x <- xs, mod x p > 0])
```

   Evaluer les expressions suivantes et commenter les résultats obtenus :

```
> primes    ==> ?
> take 10 primes    ==> ?
> takeWhile (<= 100) primes    ==> ?
> length (takeWhile (<= 10000) primes)    ==>  ?
```

Partie traitée lors du CM 04

- On souhaite définir la fonction `(pfactors n)` qui à un entier `n` associe la liste de ses facteurs premiers.

```
> pfactors 8   ==>  [2,2,2]
> pfactors 72   ==>  [2,2,2,3,3]
> pfactors 924   ==>  [2,2,3,7,11]
```

  Pour cela, compléter la définition ci-dessous :

```
pfactors :: ? -> ?
pfactors n = pfactors' n primes
   where pfactors' x (p:ps)
            | p > x          = ?
            | mod x p == 0   = ?
            | otherwise      = ?
```

- Définir la fonction `(prep xs)` qui à une liste de facteurs premiers `xs` associe sa décomposition. *(On pourra utiliser les fonctions* `takeWhile` *et* `dropWhile`*).*

```
> prep [2,2,2]    ==>  [(2,3)]
> prep [2,2,2,3,3]   ==>  [(2,3),(3,2)]
> prep [2,2,3,7,11]    ==>  [(2,2),(3,1),(7,1),(11,1)]
```

- En déduire la fonction `int2rep` qui détermine la décomposition associée à un entier.

```
> int2rep 8    ==>  [(2,3)]
> int2rep 72    ==>  [(2,3),(3,2)]
> int2rep 924    ==>  [(2,2),(3,1),(7,1),(11,1)]
```

- Pour terminer, vérifier que :
  - `(rep2int . int2rep)` est bien la fonction identité sur les entiers naturels,
  - `(int2rep . rep2int)` est bien la fonction identité sur les décompositions.

```
--
pfactors :: Int -> [Int]
pfactors n = pfactors' n primes
   where pfactors' x (p:ps)
            | p > x          = []
            | mod x p == 0   = p : (pfactors' (div x p) (p:ps))
            | otherwise      = pfactors' x ps

--
prep :: [Int] -> Decomposition
prep [] = []
prep (x:xs) = (x, (length (takeWhile (==x) (x:xs))) ) : (prep (dropWhile (==x) xs))

--
int2rep :: Int -> Decomposition
int2rep = prep . pfactors
```