TD 5 : Listes – Part 2

NB : quelques exercices de ce TD proviennent du site `https://wiki.haskell.org/H-99:_Ninety-Nine_Haskell_Problems` (qui est fortement recommandé)

1 Gray code

Le code de Gray de n bits est une séquence de tous les nombres binaires de n chiffres avec la propriété suivante : chaque deux nombres consécutifs dans la liste ne différent que dans une position.

Par exemple, le code de Gray pour $n = 4$ est
$0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000$.

Le premier élément du code est toujours $00\ldots0$ et le dernier est toujours $100\ldots0$. En utilisant cette propriété, une façon de construire le code pour n bits est de construire deux copies du code pour $n-1$ bits, inverser la deuxième copie, mettre 0 au début de chaque élément de la première copie, 1 au début de chaque élément de la deuxième copie, et prendre la concaténation.

1. Donner une implémentation d'une fonction récursive `gray :: Int ->[String]` qui produit le code de Gray, en utilisant la compréhension de listes.
2. Donner une implémentation qui utilise `map`.
3. Donner une implémentation qui utilise `foldr`.

Solution :

```
gray :: Integer -> [ String ]
gray 0 = [""]
gray n = map ('0':) (gray (n-1))  ++ (reverse $ map ('1':) $ gray (n-1))

gray2 :: Integer -> [ String ]
gray2 0 = [""]
gray2 n = [ '0':x | x <- prev ] ++ [ '1':x | x <-reverse prev ]
   where prev = gray2 (n-1)

gray3 :: Integer -> [ String ]
gray3 0 = [""]
gray3 n = foldr (\x acc -> ['0':x] ++ acc ++ ['1':x]) [] (gray3 (n-1))
```

2 Run-length Encoding

On est donné une liste avec beaucoup de doublons. On aimerait produire une version compressée de cette liste qui, à la place de chaque séquence d'éléments identiques, contient une paire de la forme `(element, frequency)`.

Exemple :

```
*Main> rle "aabbbacaad"
[('a',2),('b',3),('a',1),('c',1),('a',2),('d',1)]
```

Naturellement, il nous faut aussi une fonction qui fait la conversion inverse :

```
*Main> unrle $ rle "aabbbacaad"
"aabbbacaad"
```

1. Écrire deux implémentations récursives pour les fonctions `rle, unrle`
2. Écrire une implémentation de `rle` qui utilise `foldr`
3. Écrire une implémentation de `unrle` qui utilise `foldr`
4. Écrire une implémentation de `unrle` qui utilise `foldl`
5. Écrire une implémentation de `unrle` qui utilise la compréhension de listes et `concatMap`.

NB : les fonctions `fst :: (a,b)->a` et `snd :: (a,b)->b` retournent respectivement le premier et le deuxième élément d'une paire.

Solution :

```
rle :: Eq a => [a] -> [ (a,Int) ]
rle [] = []
rle [x] = [ (x,1) ]
rle (x:y:xs)
 | x==y = let (f1,f2)=head res2 in (f1,f2+1) : tail res2
 | otherwise = (x,1):res2
 where
   res2 = rle (y:xs)

unrle :: [ (a,Int) ] -> [a]
unrle [] = []
unrle ((x,f):xs) = (take f (repeat x)) ++ (unrle xs)

rle2 :: Eq a => [a] -> [(a,Int) ]
rle2 xs = foldr (\x acc -> if ((acc==[]) || fst (head acc)/= x)
                           then (x,1):acc
                           else ((x,1+snd(head acc)):tail acc)) [] xs

unrle2 :: [(a,Int)] -> [a]
unrle2 = foldr (\(x,f) acc -> (take f (repeat x)) ++ acc ) []

unrle3 :: [(a,Int)] -> [a]
unrle3 = foldl (\acc (x,f) -> acc ++ (take f (repeat x))) []

unrle4 :: [(a,Int)] -> [a]
unrle4 xs = concatMap (\(x,f) -> take f (repeat x)) xs
```

3 List Frequency

Donner une fonction `listFreq` qui, étant donné une liste, retourne une deuxième liste qui signifie pour chaque élément de la liste donnée, sa fréquence.

Exemple :

```
*Main> listFreq "aabaabacb"
[('a',5),('b',3),('c',1)]
```

1. Donner une implémentation qui utilise la compréhension de listes

2. Donner une implémentation qui utilise `filter`

Solution :

```
listFreq :: Eq a => [a] -> [(a,Int)]
listFreq [] = []
listFreq (x:xs) = (x,f): listFreq  (filter (/=x) xs)
          where f = 1+ (length $ filter (==x) xs)

listFreq2 :: Eq a => [a] -> [(a,Int)]
listFreq2 [] = []
listFreq2 (x:xs) = (x,f): listFreq2  [ y | y<-xs, y/=x ]
          where f = 1+ (length [ y | y<-xs, y==x ])
```

4 Permutations

Donner une fonction qui, étant donné une liste qui contient des éléments distincts, retourne une liste de toutes ses permutations.

Exemple :

```
*Main> allperms "abc"
["abc","bac","bca","acb","cab","cba"]
```

Solution :

```
allperms :: [a] -> [[a]]
allperms [x] = [[x]]
allperms (x:xs) = concatMap (puteverywhere x) (allperms xs)

puteverywhere :: a -> [a] -> [[a]]
puteverywhere x [] = [[x]]
puteverywhere x (y:ys) = (x:y:ys): map (y:) (puteverywhere x ys)
```