TD 6 : Opérations de listes et complexité

NB : quelques exercices de ce TD proviennent du livre "Algorithm Design with Haskell" de Richard Bird et Jeremy Gibbons (qui est fortement recommandé)

1 La Complexité de la Concaténation

L'objectif de cet exercice est d'étudier de plus près la complexité de l'opération ++ et l'impact de l'utilisation de `foldr` ou `foldl`. Nous commençons avec une réimplémentation de ++ et puis on l'utilise pour réimplémenter la fonction `concat`.

1. Donner une implémentation de la fonction ++.
2. Donner une implémentation de la fonction ++ en utilisant `foldr`.
3. Si on applique l'opération ++ sur deux listes `xs,ys` de taille n, m respectivement, quelle est sa complexité en tant que fonction de n, m ? Vous pouvez supposer que l'opération `(:)` est de complexité $O(1)$.
4. Donner une implémentation de la fonction `concat` en utilisant ++ et `foldr`.
5. Donner une implémentation de la fonction `concat` en utilisant ++ et `foldl`.
6. Quelle est la complexité de chacune de vos implémentations ? Laquelle est préférable ?

Solution :

```
myconcat2 :: [a] -> [a] -> [a]
myconcat2 [] xs = xs
myconcat2 (x:xs) ys = x:myconcat2 xs ys

myconcat3 :: [a] -> [a] -> [a]
myconcat3 xs ys = foldr (:) ys xs

concatr :: [[a]] -> [a]
concatr xs = foldr myconcat3 [] xs

concatl :: [[a]] -> [a]
concatl xs = foldl myconcat3 [] xs

--Tests
*Main> length $ concatl $ take (10^3) $ repeat [1..10^3]
1000000
(11.77 secs, 38,752,967,480 bytes)
*Main> length $ concatr $ take (10^3) $ repeat [1..10^3]
1000000
(0.04 secs, 56,253,744 bytes)
*Main> length $ concatr $ take (10^4) $ repeat [1..10^4]
100000000
(1.69 secs, 5,601,913,488 bytes)
```

```
*Main> length $ concatr $ take (10^5) $ repeat [1..10^2]
10000000
(0.19 secs, 571,281,496 bytes)
```

La complexité de ++ est de $O(n)$ où n est la taille de la première liste. Si on est donné une liste de listes de tailles $n_1, n_2, \ldots, n_k$, l'implémentation avec `foldr` a coût $O(n_{k-1})+O(n_{k-2})+\ldots+O(n_1) = O(\sum_{i=1}^{k-1} n_i)$, ce qui est une complexité linéaire (dans la taille du résultat). L'implémentation avec `foldl` a complexité $O(n_1) + O(n_1 + n_2) + O(n_1 + n_2 + n_3) + \ldots = O(\sum_{i=1}^{k-1}(k-i)n_i)$. Quand on a beaucoup de petites listes ($n_i = O(1)$) cela devient $O(k^2)$, c'est-à-dire une complexité quadratique. La différence est claire dans les exemples d'exécution ci-dessus, l'implémentation avec `foldr` est meilleure.

2 Steep Lists

On dira qu'une liste d'entiers est "steep" (raide) si on a la propriété suivante : pour chaque élément x de la liste, x est strictement supérieur à la somme de tous les éléments suivants. Par exemple $[4, 2, 1]$ et $[23, 11, 8, 2]$ sont steep, alors que $[5, 4, 2]$ et $[125, 15, 11, 5]$ ne le sont pas.

1. Programmer une fonction qui teste si une liste donnée est steep.
2. En utilisant `foldr`, programmer une version améliorée de votre fonction de complexité $O(n)$, où n est le nombre d'éléments de la liste. (Vous pouvez supposer que les opérations arithmétiques ont complexité $O(1)$).

Solution :

```
steep :: (Num a, Ord a) => [a] -> Bool
steep [] = True
steep (x:xs) = x>sum xs && steep xs

steep2 :: (Num a, Ord a) => [a] -> Bool
steep2 xs =  fst ( foldr (\x (b,s) -> if (b && (x>s))
                      then (True,x+s)
                      else (False,s)) (True,0) xs)
```

La première implémentation a complexité quadratique, car on recalcule la somme à chaque fois. L'implémentation avec `foldr` applique à chaque étape une fonction qui prend $O(1)$, donc sa complexité est $O(n)$.

Exemples d'exécution :

```
*Main> steep $ map (2^) [10000,9999..1]
True
(11.39 secs, 48,093,490,352 bytes)
*Main> steep2 $ map (2^) [10000,9999..1]
True
(0.10 secs, 70,731,944 bytes)
```

3 Foldr vs Foldl

Dans cet exercice on va comparer un autre aspect de `foldr` et `foldl` : parfois les deux fonctions retournent le même résultat pour des listes finies mais ne sont pas équivalentes quand on passe comme argument une liste infinie. L'exemple qu'on va utiliser est la fonction `takeWhile`.

1. Donner une implémentation de la fonction `takeWhile` en utilisant `foldr`. Quelle est la complexité de votre implémentation ?
2. Donner une implémentation de `takeWhile` en utilisant `foldl`. Quelle est la complexité de votre implémentation ?

3. Que se passe-t-il si on passe comme argument une liste infinie ?

Solution :

```
-- Correct avec foldr
mytakeWhile :: (a->Bool) -> [a] -> [a]
mytakeWhile f xs = foldr (\x ys -> if f x then x:ys else []) [] xs

-- Incorrect. Actually implements filter??
mytakeWhile2 :: (a->Bool) -> [a] -> [a]
mytakeWhile2 f xs = foldl (\ys x -> if f x then ys++[x] else ys) [] xs

-- Correct avec foldl
mytakeWhile3 :: (a->Bool) -> [a] -> [a]
mytakeWhile3 f xs = fst ( foldl (\ (ys,b) x -> if b && f x
                                               then (ys++[x],b)
                                               else (ys,False)) ([],True) xs)
```

L'implémentation avec `foldr` a complexité linéaire ($O(n)$). L'implémentation avec `foldl` est plus compliquée et a complexité $O(n^2)$. En plus, l'implémentation avec `foldr` peut fonctionner pour une liste infinie, alors que pour `foldl` ça donne toujours une boucle infinie.

```
*Main> length $ mytakeWhile (<10^8) [1..10^4]
10000
(0.03 secs, 2,310,624 bytes)
*Main> length $ mytakeWhile (<10^8) [1..10^5]
100000
(0.06 secs, 22,467,264 bytes)
*Main> length $ mytakeWhile3 (<10^8) [1..10^4]
10000
(1.72 secs, 4,297,375,520 bytes)
*Main> length $ mytakeWhile3 (<10^8) [1..2*10^4]
20000
(18.43 secs, 17,517,357,640 bytes)
*Main> mytakeWhile (<10) [1..]
[1,2,3,4,5,6,7,8,9]
(0.01 secs, 81,320 bytes)
*Main> mytakeWhile3 (<10) [1..]
^CInterrupted. -- Boucle infinie
```