TD 2 : Algorithmes de tri

1 Quicksort – Tri rapide

Pendant le cours on a vu l'implémentation de l'algorithme de tri rapide en Haskell :

```
qsort [] = []
qsort (x:xs) = (qsort left) ++ [x] ++ (qsort right)
  where
    left = onlySmaller x xs
    right = onlyLarger x xs

onlySmaller i [] = []
onlySmaller i (x:xs)
  | x < i = x: (onlySmaller i xs)
  | otherwise = onlySmaller i xs

onlyLarger i [] = []
onlyLarger i (x:xs)
  | x > i = x: (onlyLarger i xs)
  | otherwise = onlyLarger i xs
```

Cette implémentation est correcte seulement si la liste donnée ne contient pas de doublons (pourquoi ?). Donner une implémentation qui fonctionne correctement dans tous les cas.

NB : n'oubliez pas de donner les types de toutes vos fonctions.

Solution :

Il faut s'assurer qu'on ne supprime pas les doublons de `x`, donc on modifie légérement la définition de `onlySmaller`.

```
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = (qsort left) ++ [x] ++ (qsort right)
  where
    left = onlySmaller x xs
    right = onlyLarger x xs

onlySmaller :: Ord a => a -> [a] -> [a]
onlySmaller i [] = []
onlySmaller i (x:xs)
  | x <= i = x: (onlySmaller i xs)
  | otherwise = onlySmaller i xs

onlyLarger :: Ord a => a -> [a] -> [a]
onlyLarger i [] = []
onlyLarger i (x:xs)
  | x > i = x: (onlyLarger i xs)
  | otherwise = onlyLarger i xs
```

2 Mergesort – Tri fusion

Rappel : L'algorithme de tri fusion, étant donné un tableau $A[1 \dots n]$ fait les étapes suivantes :

1. Trie la première moitié de A (appel récursif)
2. Trie la deuxième moitié de A (appel récursif)
3. Fusionne les deux tableaux triés

Donner une implémentation de cet algorithme en Haskell. Vous pouvez utiliser les fonctions `take, drop`.

Solution :

```
merge :: Ord a => [a] -> [a] -> [a]
merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys)
  | x<y = x: (merge xs (y:ys))
  | otherwise = y: (merge (x:xs) ys)

mergesort :: Ord a => [a] -> [a]
mergesort [] = []
mergesort [x] = [x] -- Q: Pourquoi a-t-on besoin de cette ligne ?
mergesort xs = merge a1 a2
  where
   a1 = mergesort (take (div (length xs) 2) xs)
   a2 = mergesort (drop (div (length xs) 2) xs)
```

3 InsertionSort – Tri par insertion

Rappel : l'algorithme de tri par insertion trie un tableau $A[1 \dots n]$ comme suit :

1. On trie le tableau $A[2 \dots n-1]$.
2. On insère l'élément $A[1]$ dans la bonne position dans le tableau trié.

Donner une implémentation de cet algorithme en Haskell.

Solution :

```
insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys)
 | x<y = (x:y:ys)
 | otherwise = y:(insert x ys)

insertionsort :: Ord a => [a] -> [a]
insertionsort [] = []
insertionsort [x] = [x]
insertionsort (x:xs) = insert x ys
  where ys = insertionsort xs
```

4 SelectionSort – Tri par sélection

Rappel : l'algorithme de tri par sélection trie un tableau $A[1 \dots n]$ comme suit :

1. Répète n fois
   — On trouve l'élément minimum parmi les éléments qui ne sont pas encore triés et le retire du tableau.

Donner une implémentation de cet algorithme en Haskell. Hint : Tout d'abord, il faut écrire une fonction `findmin` qui, étant donné une liste, retourne la position de l'élément minimum de la liste. Exemple : `findmin [17,15,11,23] --> 2`.

Solution :

```
findmin :: Ord a => [a] -> Int
findmin [x] = 0
findmin (x:xs)
  | (x<y) = 0
  | otherwise = 1+findmin xs
  where y=xs!!(findmin xs)

remove :: [a] -> Int -> [a]
remove [] _ = []
remove (x:xs) 0 = xs
remove (x:xs) i = x:(remove xs (i-1))

selectionsort :: Ord a  => [a] -> [a]
selectionsort [] = []
selectionsort [x] = [x]
selectionsort xs = y:ys
  where y = xs!!(findmin xs)
        ys = selectionsort (remove xs (findmin xs))
```