Un peu de tris

I. Tri par insertion

Commençons par un tri dit *naïf*, qui est en général moins efficace que d'autres algorithmes que nous présenterons ensuite.

Question 1 Écrire une fonction

```
inserer : 'a -> 'a list -> 'a list
```

qui insert un élément à la bonne place dans une liste ordonnée par ordre croissant.

On doit avoir, par exemple :

```
# inserer 5 [0; 2; 4; 6; 8] ;;
- : int list = [0; 2; 4; 5; 6; 8]
```

Question 2 Écrire une fonction

```
tri_insertion : 'a list -> 'a list
```

qui renvoie la version triée de la liste passée en argument, en procédant de la façon suivante : on insère chaque nouvel élément à la bonne place dans ce qui est déjà trié.

Question 3 Quelle est la complexité de la fonction `tri_insertion` dans le pire des cas ?

II. Tri fusion

Le principe du tri fusion est très simple : si l'on a une liste à trier, on la coupe en deux, on trie chaque moitié et on fusionne les sous-listes triées. La magie de la programmation récursive fait que si on sait découper une liste en deux et fusionner deux listes triées, alors on sait trier une liste.

Question 4 Écrire une fonction

```
fusionner: 'a list -> 'a list -> 'a list
```

qui fusionne deux listes triées par ordre croissant.

Par exemple, on doit avoir :

```
# fusionner [2; 4; 5; 8] [0; 1; 3; 4; 9] ;;
- : int list = [0; 1; 2; 3; 4; 4; 5; 8; 9]
```

Question 5 Écrire une fonction

```
decomposer: 'a list -> 'a list * 'a list
```

qui, étant donné une liste, renvoie les listes composées des éléments de positions paires et impaires de la liste initiale. Ainsi, on doit avoir :

```
# decomposer [4; 3; 1; 7; 6; 5; 9; 2] ;;
- : int list * int list = ([4; 1; 6; 9], [3; 7; 5; 2])
```

Question 6 En déduire une fonction

```
tri_fusion: 'a list -> 'a list
```

qui trie une liste selon l'algorithme du tri fusion décrit précédemment. On pourra distinguer le cas où la liste à trier contient au plus un élément (dans ce cas, que faut-il faire ?) et où elle contient au moins deux éléments.

On verra bientôt que la complexité dans le pire des cas est en $O(n \ln n)$ en la longueur n de la liste, et nous verrons un peu plus tard que c'est optimal.

III. Tri rapide

Pour finir, un dernier algorithme de tri, dit *rapide* (ou *quicksort*). L'idée est la suivante. Supposons que l'on veut trier la liste :

```
[4; 3; 1; 7; 6; 5; 9; 2]
```

on choisit un élément (en général le premier, ici `4`) que l'on appelle le *pivot* et on partitionne la liste (privée du pivot) en deux : celle des éléments plus inférieurs (ou égaux) au pivot, et celle des éléments strictement supérieurs. Si on sait trier les sous-listes, en remettant tout dans le bon ordre, on a la liste de départ triée.

Question 7 Écrire une fonction

```
partitionner: 'a * 'a list -> 'a list * 'a list
```

qui, étant donné une valeur pivot et une liste retourner la paire composée de la liste des éléments plus petits que le pivot et la liste des éléments plus grands que le pivot.

Ainsi, on veut avoir :

```
# partitionner 4 [3; 1; 7; 6; 5; 9; 2] ;;
- : int list * int list = ([3; 1; 2], [7; 6; 5; 9])
```

Question 8 En déduire une fonction

```
tri_rapide: 'a list -> 'a list
```

qui renvoie la liste donnée en argument triée selon l'algorithme *quicksort*.

Dans l'exemple précédent, `[3; 1; 2]` devient `[1; 2; 3]` et `[7; 6; 5; 9]` devient `[5; 6; 7; 9]` et donc en remettant les morceaux dans le bon ordre, on obtient

```
[1; 2; 3; 4; 5; 6; 7; 9] = [1; 2; 3] @ (4 :: [5; 6; 7; 9])
```

On rappelle que la concaténation de deux listes se fait à l'aide de l'opérateur infixe `@`.

La complexité dans le pire des cas de *quicksort* est en $O(n^2)$, mais en moyenne, on a bien du $O(n \ln n)$, et en moyenne à nouveau, il est plus rapide que le tri fusion.

On aura l'occasion de revenir sur divers caractéristiques de ces algorithmes de tri.

IV. Testons un peu ces fonctions

Question 9 Écrire une fonction

```
est_croissante: 'a list -> bool
```

qui indique si la liste passée en argument est croissante. Normalement, toutes les fonctions retournées par nos fonctions de tri doivent être croissantes.

Nous allons maintenant engendrer aléatoirement des listes à classer. Pour cela, nous avons besoin d'un peu de mise en place. Commençons par initialiser le générateur de nombres aléatoires :

```
Random.self_init() ;;
```

Voici maintenant une fonction permettant de générer une liste de flottants aléatoires d'une longueur spécifiée :

```
let rec liste_aleatoire n =
  if n <= 0 then
    []
  else
    (Random.float 1.) :: liste_aleatoire (n - 1)
;;
```

Question 10 Engendrer une liste aléatoire `l` de longueur 1000 et tester que les

fonctions `tri_insertion`, `tri_fusion` et `tri_rapide` appliquées à `l` renvoient bien des listes croissantes.

Enfin, une fonction pour chronométrer le temps d'exécution d'une autre fonction :

```
let chrono f x =
  let t1 = Sys.time () in
  let _ = f x in
  let t2 = Sys.time () in
  t2 -. t1
;;
```

Question 11 Chronométrer le temps mis pour trier une liste aléatoire de 100 éléments selon les trois algorithmes. Faire de même pour 1 000 éléments, pour 10 000 éléments.

Remarques

- ⋆ Normalement, vous devez voir la différence entre un tri de complexité quadratique (le tri par insertion) à un tri en $n \ln n$ comme le tri fusion ou le tri rapide.
- ⋆ Il est fort possible que vous ne voyez pas de réels différences entre ces deux derniers en temps d'exécution. C'est normal, puisque le « vrai » *quicksort* s'applique en fait à des tableaux.

En étudiant le fonctionnement de ces algorithmes, si l'on utilise une liste déjà triée par ordre croissant, le tri par insertion devient de complexité linéaire (c'est un tri *adaptatif*) alors que le tri rapide est de complexité quadratique.

Question 12 Mettre cela en évidence.

V. Pour ceux qui en veulent plus

Question 13 Pour le tri par insertion, pouvez-vous faire `inserer` récursive terminale ? Pouvez-vous faire `tri_insertion` en utilisant `List.fold_right` voire, encore mieux, `List.fold_left` ?

Amélioration du tri fusion

Si le tri fusion a une complexité optimale, il demeure un problème. Pour voir cela, essayez de trier une liste de 1 000 000 éléments. Vous devriez avoir une erreur du type :

```
Stack overflow during evaluation (looping recursion?).
```

Pour améliorer cela, il faut passer en récursion terminale.

Question 14 Pouvez-vous écrire une version récursive terminale de `fusionner` ?

Petit indice, on peut construire la liste fusionner « à l'envers », décroissante, de façon récursive terminale puis de retourner le tout grâce à la fonction `List.rev` ou plus généralement :

```
List.rev_append: 'a list -> 'a list -> 'a list.
```

Le code de cette fonction est le suivant :

```
let rec rev_append l1 l2 =
  match l1 with
  | [] -> l2
  | a :: l1' -> rev_append l1' (a :: l2)
;;
```

On a par exemple :

```
# List.rev_append [1; 2; 3] [4; 5] ;;
- : int list = [3; 2; 1; 4; 5]
```

Question 15 Et pouvez-vous rendre la fonction `decomposer` récursive terminale (sachant que l'ordre dans les sous-listes importe peu) ?

Question 16 En déduire une version totalement récursive terminale du tri fusion. Normalement, vous devez pouvoir trier une liste de 100 000 éléments maintenant (et même de 1 000 000 voire de 10 000 000 éléments même si cela commence à prendre beaucoup de temps).

Amélioration du tri rapide

On peut aussi modifier le programme de tri rapide précédent pour le rendre récursif terminal.

Question 17 Modifier la fonction `partitionner` pour la rendre récursive terminale. On pourra utiliser deux arguments supplémentaires correspondant aux listes (correspondant aux plus petits et plus grands éléments) en train d'être construites.