TD 6
Tris

Le but du TD est de travailler sur différents types de tris de listes ou d'un tableau d'entiers. Nous avons déjà vu le tri par sélection, dont la complexité en d'ordre $O(n^2)$, ainsi que le tri fusion, dont la complexité en d'ordre $O(n\ln(n))$.

Exercice 1 :

1. Écrire une fonction `liste_aleatoire` qui prend en entrée un entier n et renvoie une liste de nombres aléatoires de taille n inférieur à 1000.

   ```
   liste_aleatoire : int -> int list
   ```

2. Écrire une fonction `tableau_aleatoir` qui prend en entrée un entier n et renvoie un tableau de nombres aléatoires de taille n inférieur à 1000.

   ```
   tableau_aleatoire : int -> int array
   ```

Exercice 2 : Le tri par insertion est un autre algorithme que l'on peut qualifier de naïf. Cet algorithme consiste à piocher une à une les valeurs du tableau et à les insérer, au bon endroit, dans le tableau trié constitué des valeurs précédemment piochées et triées. Les valeurs sont piochées dans l'ordre où elles apparaissent dans le tableau. Soit p l'indice de la valeur piochée, les $(p-1)$ premières valeurs du tableau constituent le tableau trié dans lequel va être inséré la p-ième valeur.

1. Tri par insertion dans un tableau.
   (a) Écrire la fonction `tri_inser_tab` qui trie un tableau donné.

   ```
   tri_inser_tab : 'a array -> 'a array
   ```

   (b) Calculer la complexité de la fonction dans le pire des cas.
2. Tri par insertion dans une liste.
   (a) Écrire la fonction `insere` qui insère un élément donné dans une liste (supposée triée).

   ```
   insere : 'a -> 'a list -> 'a list
   ```

   (b) Écrire la fonction `tri_insertion` en utilisant la fonction précédent :

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

Exercice 3 : Le **tri à bulles** ou **tri par propagation** est un algorithme de tri qui consiste à faire remonter progressivement les plus grands éléments d'une liste, comme les bulles d'air remontent à la surface d'un liquide. L'algorithme parcourt la liste, et compare les couples d'éléments successifs. Lorsque deux éléments successifs ne sont pas dans l'ordre croissant, ils sont échangés. Après chaque parcours complet de la liste, l'algorithme recommence l'opération. Lorsqu'aucun échange n'a lieu pendant un parcours, cela signifie que la liste est triée : l'algorithme peut s'arrêter.

1. La fonction `parcours` fait un passage sur toute la liste en échangeant progressivement les éléments.
   Tester cette fonction à la main sur la liste $[5;1;6;4;3]$.

2. Démontrer que à l'issue de la fonction `parcours`, la valeur maximum de la liste se retrouve en dernière position.

3. Écrire la fonction `parcours` qui procède au parcours d'une liste, renvoie la liste modifiée et le booléen correspondant au fait d'un échange.

   ```
   parcours : 'a list -> 'a list * bool
   ```

4. Écrire la fonction `tri_bulle`. Le programme s'arrêtera si aucune permutation ne s'est produite.

   ```
   tri_bulle : 'a list -> 'a list
   ```

5. Calculer le nombre d'appel récursif dans le meilleur et dans le pire des cas.

Exercice 4 : **Tri rapide (quicksort)**

La méthode consiste à placer un élément d'un tableau d'éléments à trier (appelé pivot) à sa place définitive en permutant tous les éléments de telle sorte que tous ceux qui lui sont inférieurs soient à sa gauche et que tous ceux qui lui sont supérieurs soient à sa droite. Cette opération s'appelle le partitionnement. Pour chacun des sous-tableaux, on définit un nouveau pivot et on répète l'opération de partitionnement. Ce processus est répété récursivement, jusqu'à ce que l'ensemble des éléments soit trié.

1. Écrire la fonction `partition` qui renvoie pour une liste et un pivot donnés en couple, deux listes dont tous les éléments de la première sont inférieurs ou égaux au pivot et tous les éléments de la deuxième sont supérieurs au pivot :

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

2. Écrire la fonction `tri_rapide` en prenant comme pivot le premier élément de la liste :

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

3. On note $C(n)$ le nombre de comparaison nécessaire dans le tri rapide pour une liste de taille n. On a donc $C(0) = C(1) = 0$.

   (a) Déterminer la complexité dans le pire des cas.

   (b) Donner un exemple d'une liste de taille 7 corresp ondant à la complexité dans le meilleur des cas.

   (c) On cherche à estimer le nombre de comparaison nécessaire en moyenne dans le tri rapide.
   On note $C_m(n)$ ce nombre pour une liste de taille n.
   On suppose que la liste de taille n est composée des n entiers distincts de $[1, n]$.

      i. Montrer que, pour tout $n \geq 2$:

      $$C_m(n) = n - 1 + \frac{2}{n} \sum_{k=0}^{n-1} C_m(k)$$

      ii. En déduire que, pour $n \geq 2$:

      $$nC_m(n) = 2(n-1) + (n+1)C_m(n-1)$$

iii. On pose, pour $n \geq 0$, $D(n) = \dfrac{C_m(n)}{n+1}$, montrer que, pour $n \geq 2$:

$$D(n) = 2\frac{n-1}{n(n+1)} + D(n-1)$$

iv. En déduire la relation :

$$D(n) = 2\sum_{k=1}^{n} \frac{k-1}{k(k+1)}$$

puis

$$D(n) = -\frac{2n}{n+1} + 2\sum_{k=1}^{n} \frac{1}{k+1}$$

v. Sachant que $\displaystyle\sum_{k=1}^{n} \frac{1}{k} \sim \ln(n)$, en déduire un ordre de grandeur de la complexité en moyenne du tri rapide.