TD 2 (algorithmique) - Listes.

Dans les exercices qui suivent, vous pouvez supposer que tous les éléments d'une liste sont comparables, et que les types des données de deux listes différentes sont compatibles.

Exercice 1

Écrivez une fonction `insertion_triee(T, x)` qui insère, en $O(|T|)$, un élément `x` dans une liste triée `T` de façon à ce que `T` reste triée après l'insertion. Attention, `T` peut être vide.

Indices

Plusieurs options :

1. chercher le bon endroit où insérer `x`, ajouter un nouvel élément bidon en fin de liste, puis effectuer les décalages de manière à pouvoir insérer `x`. Attention, il faut bien veiller, lorsqu'on fait les décalages, à copier les éléments comme suit : `T[j] = T[j - 1]` en faisant décroître la valeur de `j` ; si on le fait dans l'autre sens, on perdra des informations.
2. rajouter `x` en fin de liste, puis à faire des échanges adjacents successifs en descendant tant que `x` est inférieur à son prédécesseur. Cette solution est plus simple, mais moins efficace (pourquoi ?).
3. on pourrait aussi, dans l'option 1, se servir de la dichotomie pour rechercher l'endroit où réaliser les insertions. Cela n'affecterait pas la complexité (pourquoi ?), mais on pourrait remarquer des améliorations en pratique.

Exercice 2

Une permutation de n éléments est une liste contenant les entiers de 1 à n une et une seule fois. Écrivez une fonction qui vérifie, en $O(n)$, si une liste donnée est bien une permutation.

Exercice 3

Écrivez une fonction qui, étant donnée une liste de n entiers, déplace tous les 0 vers la fin de la liste. Il n'est pas nécessaire de préserver l'ordre des autres éléments. Par exemple, `[0, 1, 0, 2, 3]` sera transformée en `[3, 1, 2, 0, 0]`. Votre solution doit être en $O(n)$.

Indices

Extraire des zéros et les réinsérer en fin de liste fonctionne mais ne donne pas la bonne complexité (pourquoi ? quelle complexité obtient-on ?). Pour effectuer les déplacements en temps constant, on va échanger les éléments en "mauvaise place" : un 0 situé avant un élément non nul, et un élément non nul situé avant un 0.
Pour être sûr de n'échanger que des éléments mal placés, on va maintenir un curseur `p` donnant l'indice du dernier élément non nul de la liste. Il suffira ensuite de parcourir la liste de gauche à droite avec un autre indice `i`, en échangeant les éléments en positions `i` et `p` uniquement si l'élément en position `i` est un 0. Attention, il faut bien entendu mettre à jour la position `p` à chaque étape.
Pour prouver que la fonction est en $O(n)$, il suffit de remarquer que le nombre d'incrémentations de i ainsi que le nombre de décrémentations de p sont inférieurs ou égaux à n.

Exercice 4

Modifiez la solution de l'exercice précédent pour que l'algorithme préserve l'ordre des éléments non nuls (c'est-à-dire que `[0, 1, 0, 2, 3]` sera transformée en `[1, 2, 3, 0, 0]` et pas `[3, 1, 2, 0, 0]`). Votre solution doit être en $O(n)$.

Exercice 5

Soit `S` et `T` deux listes triées. Écrivez une fonction `fusion(S, T)` qui renvoie une liste triée contenant tous les éléments de `S` et de `T` (les doublons doivent être préservés). Votre fonction doit être en $O(|S| + |T|)$.

Exercice 6

Écrivez une fonction `reordonner(T)` qui réorganise la liste `T` de n éléments en plaçant tous les éléments inférieurs ou égaux au premier élément avant lui. Par exemple, si `T = [3, 1, 4, 2, 5, 2]`, alors `[1, 2, 2, 3, 4, 5]` est un résultat valide qu'on pourrait obtenir après avoir appelé la fonction. Il n'est pas nécessaire de préserver l'ordre des autres éléments, mais la fonction doit être en $O(n)$.