EA4 – Éléments d'algorithmique
TP n° 3 : tris (première partie)

Modalités de rendu : Le TP 3 est à rendre sur Moodle en fin de TP dans un unique fichier `tp3.py`. Vous avez ensuite quelques jours supplémentaires pour le compléter (date à voir avec votre chargé de TP).

Le but de ce TP est d'écrire les implémentations les plus efficaces possibles des tris vus en cours. Pour cela, nous vous fournissons dans le fichier :

- une fonction `mesure` qui prend en paramètre un algorithme de tri et une liste, et renvoie le temps mis par cet algorithme pour trier cette liste ;
- une fonction `mesureMoyenne` qui prend en paramètre un algorithme de tri et un tableau de listes, et renvoie le temps moyen mis par cet algorithme pour trier une liste de ce tableau ;
- une fonction `courbes` qui trace les courbes de temps pour des algorithmes et des tableaux de listes donnés en paramètre ;
- une fonction `affiche` qui affiche les courbes précédemment tracées ;
- une fonction `compareAlgos` qui compare tous les algorithmes implémentés.

Exercice 1 : préparation des tests

Dans cet exercice, il s'agit d'écrire différentes fonctions qui permettront par la suite de tester les fonctions de tri implémentées et de comparer leur efficacité.

1. ✍ Implémenter une fonction de tri par sélection.
2. ✍ Écrire une fonction `randomPerm` qui prend en paramètre un entier `n` et renvoie une permutation aléatoire de longueur `n`. Pour cela, vous réutiliserez le code de la fonction de tri par sélection, mais vous partirez d'un tableau contenant les entiers de `1` à `n` et au lieu de calculer, à l'étape `i`, l'indice correspondant à la valeur minimum de `T[i:]`, vous tirerez un indice aléatoire dans ce même sous-tableau. Vous pouvez utiliser la fonction `random.randint(a, b)` pour tirer aléatoirement un indice entre `a` et `b` (bornes comprises).
3. ✍ Écrire une fonction `testeQueLaFonctionTrie` qui prend en paramètre une fonction de tri `f` et l'applique sur des permutations aléatoires de taille `i` variant de 2 à 100 et vérifie à chaque fois que le résultat est un tableau comportant tous les entiers entre 1 et `i`, dans l'ordre. Elle renvoie `True` dans ce cas. Sinon elle affiche un exemple de tableau que la fonction `f` ne trie pas correctement et renvoie `False`. Attention à faire une copie du tableau avant de le trier ! Vous utiliserez `testeQueLaFonctionTrie` pour tester la fonction de tri par sélection.
4. ✍ Écrire une fonction `randomTab` qui prend en paramètre une taille `n` et deux bornes `a` et `b` et génère un tableau aléatoire de taille `n` contenant des entiers compris entre les bornes `a` et `b`.
5. ✍ Écrire une fonction `derangeUnPeu` qui prend trois arguments `n`, `k` et `rev` et effectue `k` échanges entre des positions aléatoires sur la liste des entiers de 1 à `n` si `rev` vaut `False` ou sur la liste des entiers `n` à 1 si `rev` vaut `True`.
6. Comparer le tri par sélection sur les différents types de permutation en décommentant la partie `main` du fichier fourni (partie intitulée « question 1.6 »).

Exercice 2 : implémentation des tris

1. ✍ Implémenter les trois variantes du tri par insertion vues au TD n° 4 :
   - avec échange d'éléments successifs (voir `triInsertionParLaDroite`),
   - avec insertion directe à la bonne position (exercice 2.6 du TD n° 4),
   - et avec recherche dichotomique de la position (exercices 2.4 et 2.6 du TD n° 4).

   Tester ces fonctions à l'aide de la fonction `testeQueLaFonctionTrie`.
2. ✍ Écrire une fonction de tri fusion et la tester de la même manière.
3. ✍ Écrire une fonction de tri à bulles (voir `triBulles` du TD n° 4) et la tester de même.
4. Comparer les différents tris implémentés dans cet exercice ainsi que le tri par sélection en décommentant la partie `main` du fichier fourni (partie intitulée « question 2.4 »). Commenter dans le fichier `tp3.py` les résultats obtenus.

Exercice 3 : amélioration du tri par insertion

Vous savez déjà que le tri par insertion est un tri particulièrement efficace sur des listes presque triées. Cependant les éléments initialement très mal placés se comportent comme les « tortues » du tri à bulles. Le but de cet exercice est de concevoir une amélioration du tri par insertion qui déplace rapidement chaque élément à une place proche de la sienne. Le tri Shell, que vous allez écrire dans cet exercice, est simple, mais l'étude de son temps d'exécution est complexe. Certains problèmes liés à ce sujet sont toujours ouverts. Il a une complexité de $\Theta(n \log^2(n))$ pour certaines suites de pas, mais la complexité pour la suite « empiriquement optimale » donnée ci-dessous n'est pas connue.

1. ✍ Écrire une fonction `triInsertionPartiel(T, gap, debut)` qui trie par insertion le sous-tableau `T[debut::gap]` de `T` (constitué des cases d'indice `debut`, `debut+gap`, `debut+2*gap`...) : la variable `gap` représente l'espacement entre deux cases consécutives du sous-tableau que l'on souhaite trier.
2. ✍ Écrire une fonction `triShell` qui, en utilisant la fonction de la question précédente, trie un tableau en plusieurs étapes : chaque étape réalise le tri partiel de tous les sous-tableaux correspondant à un espacement donné ; l'espacement décroît à chaque étape, le dernier espacement étant nécessairement égal à `1` pour que le tableau soit trié à la fin de l'algorithme. Vous utiliserez la suite d'espacements `701`, `301`, `132`, `57`, `23`, `10`, `4`, `1`, qui est celle qui donne les meilleurs résultats empiriques.
3. Décommenter la partie `main` pour tracer les courbes de temps pour `triShell` (partie intitulée « question 3.3 »).
4. Décommenter la fin du main afin de comparer les différents algorithmes implémentés (partie intitulée « question 3.4 »). Commenter dans le fichier `tp3.py` les résultats obtenus.

Exercice 4 : méthode sort

A partir de la version 3.11 de python 3, la méthode sort est basée sur une variante du tri par fusion connue comme *Powersort*.[1] D'après les auteurs, sur une permutation aléatoire,

1. I. Munro et S. Wild, *Nearly-optimal mergesorts : fast, practical sorting methods that optimally adapt to existing runs*, Proc. ESA, 2018.

le *Powersort* a une performance comparable à l'algorithme de tri par fusion traditionnel. Comparez la vitesse du tri par fusion et de la méthode sort sur des permutations aléatoires avec environ 10^6 éléments. Comment expliquez-vous les résultats observés ?