EA4 – Éléments d'algorithmique
TD n° 5 : permutations – tri par fusion

Exercice 1 : permutations (décomposition en cycles, produit, inverse, puissances)
On considère les permutations suivantes :

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 5 & 9 & 7 & 2 & 1 & 3 & 10 & 8 & 4 & 6 \end{pmatrix} \text{ et } \tau = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 1 & 5 & 8 & 4 & 2 & 7 & 6 & 3 & 9 & 10 \end{pmatrix}.$$

1. Écrire σ et τ en notation cyclique (c'est-à-dire comme produits de cycles disjoints).
2. Calculer les produits $\sigma\tau(= \sigma \circ \tau)$ et $\tau\sigma$.
3. Calculer les inverses σ^{-1}, τ^{-1} et $(\sigma\tau)^{-1}$.
4. Calculer σ^{2024}.

Exercice 2 : tri fusion
On rappelle le code du tri par fusion vu en cours :

```
def tri_fusion(T, debut, fin) :
  if fin - debut < 2 : return T[debut:fin]
  else :
    milieu = (debut + fin)//2
    gauche = tri_fusion(T, debut, milieu)
    droite = tri_fusion(T, milieu, fin)
    return fusion(gauche, droite)
```

1. Appliquez le à la main sur le tableau `[4,2,5,6,1,4,1,0]`.
   Comptez précisément le nombre de comparaisons effectuées.
2. Écrivez une version itérative `fusionIterative(T1, T2)` (de complexité linéaire) qui effectue la fusion de tableaux supposés déjà triés.
3. On dira qu'une *inversion* dans un tableau `T` est un couple de positions correspondant à des éléments mal ordonnés. Formellement, ce sont des couples d'indices `(u,v)` avec `u < v` et tels que `T[v] < T[u]`.
   Par exemple si `T = [1,2,8,4,5]`, les inversions ne concernent que l'élément 8 à la position `u = 2`, pour deux indices `v` possibles : ceux des éléments 4 et 5. Il y a donc 2 inversions présentes dans ce tableau.
   Écrire une fonction `nbInversionsNaive(T)` qui calcule le plus naturellement possible le nombre d'inversions présentes dans T. Quelle est sa complexité ?
4. Dans le cas particulier où deux tableaux `T1` et `T2` sont déjà triés, on peut calculer le nombre d'inversions présentes dans le tableau `T = T1 + T2` en s'appuyant sur la structure du code de la fusion.
   Reprendre la fonction `fusionIterative(T1, T2)` précédemment écrite, et la modifier pour qu'elle retourne un couple `(res, nbInv)` constitué du tableau `res` résultat de la fusion de `T1` et `T2`, et de l'entier `nbInv` décomptant les inversions présentes dans `T = T1 + T2`.
5. En déduire un programme `nbInversions(T)` basé sur le tri fusion et votre dernière fonction `fusionIterative(T1, T2)` pour calculer le nombre d'inversions d'un tableau `T` de taille n en temps $\Theta(n \log n)$.
   Indication : vous réfléchirez à ce que devient une inversion présente dans `T` lorsqu'on le découpe en deux parties égales.

Exercice 3 : montagnes (partiel 2022)

On dit qu'un tableau `T` de n entiers est une *montagne* s'il est constitué d'une première partie strictement croissante, suivie d'une deuxième strictement décroissante, chacune pouvant éventuellement être vide ; autrement dit, `T` est une montagne s'il est strictement croissant ou décroissant, ou s'il existe un certain indice $m \in [\![1, n-2]\!]$ tel que :

$$\texttt{T[0]} < \texttt{T[1]} < \ldots < \texttt{T[m]} \quad \text{et} \quad \texttt{T[m]} > \texttt{T[m+1]} > \ldots > \texttt{T[n-1]}.$$

1. Proposer un algorithme `est_une_montagne(T)` qui teste si `T` est une montagne. Justifier rapidement sa correction et sa complexité.

On suppose maintenant que `T` est une montagne.

2. Proposer un algorithme `pied(T)` le plus efficace possible qui renvoie le plus petit élément de `T`. Justifier sa correction et sa complexité.

3. Étant donné un indice i, comment tester *en temps constant* si $i < m$, où m est l'indice (inconnu *a priori*) du maximum de `T` ?

4. En déduire un algorithme `sommet(T)` le plus efficace possible qui renvoie le plus grand élément de `T`. Justifier rapidement sa correction et sa complexité.

5. Proposer un algorithme `nivelle(T)` le plus efficace possible qui renvoie un tableau trié contenant les mêmes éléments que `T`. Justifier rapidement sa correction et sa complexité.

Exercice 4 : opérations ensemblistes

On considère deux ensembles représentés par des tableaux sans doublons `E` et `F`, et on veut obtenir un tableau `I` représentant l'intersection des deux ensembles, c'est-à-dire l'ensemble des éléments qui sont à la fois dans `E` et dans `F`.
Proposer deux solutions : l'une consistant à examiner itérativement si les éléments de `E` sont dans `F`, et l'autre en commençant par trier les deux tableaux.
Comparer les complexités de ces deux algorithmes.
Même question pour l'union sans doublons.