Tri de listes

Généralités

On rappelle qu'une liste `L` est dite **triée** (sous-entendu par ordre croissant) si pour tout $i \in [\![0 ; \text{len}(L) - 2]\!]$, $L[i] \leqslant L[i+1]$.

1. Écrire une fonction `EstTriée(L)` qui, a une liste de nombres, renvoie `True` si la liste `L` est triée et `False` sinon.
2. Tester cette fonction avec divers listes, certaines triées d'autres non.
3. Écrire une fonction `ListeHasard(n,a,b)` qui renvoie une liste de longueur n dont les nombres sont choisis au hasard et uniformément dans $[\![a ; b]\!]$ (où a et b sont des entiers avec $a \leqslant b$.)

On rappelle qu'une fonction peut modifier les listes définies en dehors de la fonction (que ça soit avec des append, des pop ou des modifications d'éléments de la liste) même si on ne retourne pas la liste en question contrairement aux nombres (c'est ce qu'on appelle l'**effet de bord**), tester et comprenez le code suivant :

```
a = 3
b = 5
def f(b,a):
    a = a+b         #Modification de a dans la fonction
    b = 3*a      #Modification de b dans la fonction
    return b
print(f(b,a))     #exécution de f avec a=3 et b=5 et affichage
print(a)
print(b)

L = [3,5]
b = 5
def g(b):
    L[0] = b + L[1]         #Modification globale de L
    L.pop()                 #Modification globale de L
    L.append(8)          #Modification globale de L
    b = 3*L[0]
    L = [2,-9] + L # Modification locale de L, car création
    #d'une nouvelle liste dont le nom est L
    return b
print(g(b))#exécution de g avec b=5 et affichage résultat
print(L)
print(b)
```

Tri à bulles : glou-glou !

Le tri à bulles est un tri qui s'exécute de la façon suivante : on parcourt la liste (de gauche à droite), dès qu'on voit deux éléments consécutifs qui ne sont pas dans le bon ordre, on les échange, et on parcourt la liste tant qu'on trouve des changements à faire :

1. Commencer par créer une fonction `Changement(L)`, où `L` est une liste, qui parcourt une fois tous les indices possibles i et qui échange les valeurs de `L[i]` et de `L[i+1]` lorsque `L[i]>L[i+1]`. Cette fonction renverra `True` si elle fait au moins un échange de valeurs et `False` sinon. À Noter que d'après le rappel, cette fonction n'a pas de `return` car la liste est modifiée par effet de bord.
2. Créer une fonction `TriABulles` à un paramètre une liste, cette fonction appelle la fonction `Changement` tant que celle-ci trouve des changements à faire. À la fin, il n'y a plus de changement à faire donc la liste est triée. À Noter que d'après le rappel, cette fonction n'a pas de `return`.
3. Tester votre fonction avec une liste au hasard de longueur 20 d'entiers entre 0 et 9.

Tri par insertion : vous triez vos cartes ainsi !

Le tri par insertion trie par tranches croissantes de la liste en insérant chaque nouvel élément à sa bonne place dans la tranche déjà triée.

1. Écrire une fonction Python `InsertionElement(L,j)` qui prend en argument une liste `L` et un entier `j` (on suppose que $j \in [\![0 ; \text{len}(L) - 1]\!]$ et que la tranche `L[0:j]` est triée. La fonction `InsertionElement(L,j)` mais modifie la liste de manière à insérer `L[j]` à la bonne place dans la tranche `L[0:j+1]`. Ainsi, à l'issue de l'exécution de la fonction, la tranche `L[0:j+1]` est triée. Cette fonction modifie `L` par effet de bord et n'a donc pas de `return`.
2. À l'aide de la fonction `InsertionElement(L,j)`, écrire la fonction `TriInserction(L)` qui trie la liste `L` en la modifiant par effet de bord, cette fonction n'a donc pas de `return`.

Vous kiffez vos voisins ?

1. Écrire une fonction `PointsHasard(n,m)` qui renvoie une liste de $n+m$ éléments, tels que les n premiers éléments soit de la forme `(x,y,"b")` où x et y sont des nombres choisis uniformément dans $[\,0\,;1\,]$ et les m derniers éléments sont de la forme `(x,y,"r")` où x et y sont des nombres choisis uniformément dans $[\,0.5\,;1.5\,]$.
2. Posons `L = Points(Hasard(10,10))` et $P = (x, y)$ où x et y sont choisis uniformément dans $[\,0\,;1.5\,]$. Afficher les points de L et P tels que les n premiers points de L soit en bleus et les m derniers en rouge et le point P en vert.
3. Écrire une fonction `Distance(t,c)` où t est un triplet de la forme (x, y, col) et c est un couple de la forme (x', y') qui renvoie la distance entre les points (x, y) et (x', y') soit d'après le théorème de Pythagore :
$$\sqrt{(x-x')^2+(y-y')^2}$$
col codant la couleur du premier point.
4. À l'aide d'un tri à bulles modifié ou d'un tri par insertion modifié, trier la liste `L` par ordre croissant de la distance au point P. Les points les plus proches de P seront en premier. Il faut modifier les algorithmes de tri, car au lieu de regarder si $L[i] < L[j]$ (ce qui n'a plus de sens car $L[i]$ et $L[j]$ ne sont pas des nombres mais des triplets) il faut tester si `Distance(L[i],P)<Distance(L[j],P)`.
5. On cherche à deviner si le point P est plus probablement un point bleu ou un point rouge, pour ça on va procéder par majorité, on va regarder si parmi les k plus proches voisins de P une majorité est rouge ou bleu, écrire donc une fonction `PlusProchesVoisins(L,P,k)` qui renvoie bleu ou rouge suivant ce principe.
6. Tester avec k impair. Quel est l'avantage de prendre k impair ?

Remarque 1. Cet algorithme appelé k-plus proches proches voisins est un algorithme d'intelligence artificielle et plus précisément un algorithme d'apprentissage supervisé.

Plus stupide que ça, tu meurs !

Voici la pire méthode de tri au monde : à partir d'une liste, tant qu'elle n'est pas triée, on permute ses éléments au hasard et on regarde si la liste obtenue est triée.

1. Écrire une fonction récursive `ListePermutations(n)` qui, pour $n \in \mathbb{N}^*$, renvoie toutes la liste de toutes permutations de 0, 1 jusqu'à $n-1$, chaque permutation sera donnée sous forme d'une liste. Par exemple, pour $n = 3$, la fonction renverra la liste suivante (peu importe l'ordre de ces éléments)
```
[[0,1,2], [0,2,1], [1,2,0], [1,0,2], [2,1,0], [2,0,1]]
```
2. Écrire une fonction `Permutation(L,Per)` qui renvoie la liste des éléments de `L` mais permutés par la liste `Per` (avec `L` et `Per` de même longueur). Par exemple, si `L=[-4,8,9,1]` et `Per=[1,3,2,0]`, la fonction renverra la liste `[8,1,9,-4]`, en effet :
```
[8,1,9,-4]=[L[1],L[3],L[2],L[0]]
=[L[Per[0]],L[Per[1]],L[Per[2]],L[Per[3]]]
```
3. Coder la fonction `TriStupide` qui tri une liste par cette méthode (on utilisera la fonction `rd.choice` pour choisir une permutation au hasard).
4. Si `L` est une liste de longueur n dont les éléments sont deux à deux distincts, quelle est le nombre moyen de tentatives qu'il faudra avant que la liste soit triée ?