VII - Algorithmes de tri et applications en statistique

On considère une liste de n nombres pas forcément distincts qu'il faut trier par ordre croissant.

1) - Tri par insertion (Insertion sort)

Dans cet algorithme, on parcourt la liste à trier de l'indice 1 à l'indice $n-1$. À l'étape i, le terme d'indice i (en gris foncé dans l'exemple 1) doit être inséré à sa place parmi les termes qui précèdent (en gris clair) qui sont rangés dans l'ordre croissant.
Pour déterminer le rang k où le terme d'indice i doit s'insérer il y a deux stratégies :

1. On compare le terme à insérer avec le premier puis avec le deuxième etc, jusqu'à obtenir sa place.
2. On détermine où ce terme doit être inséré par dichotomie.

Enfin on insère au rang k le terme d'indice i avec la méthode *insert* puis on retire l'élément originel à l'indice $i+1$ par la méthode *pop*. En procédant de la sorte, les $i+1$ premiers termes seront rangés dans l'ordre croissant. On passe alors à l'étape $i+1$ et ainsi de suite.

Rappel concernant la méthode *insert* : `lst.insert(k,x)` insère l'objet x dans la liste lst au rang k.

Rappel concernant la méthode *pop* : `lst.pop(k)` retire le terme d'indice k de la liste lst et le renvoie.

Rappel concernant la recherche par dichotomie du rang d'insertion d'un objet dans une liste triée : à chaque étape on divise l'ensemble des indices admissibles (définis par deux curseurs) en deux parties d'importance équivalente puis on choisit dans quelle partie doit se trouver l'indice de l'objet que l'on doit insérer. Ce choix se fait par une comparaison entre l'objet à insérer et le terme médian. On recommence cette procédure jusqu'à trouver le rang d'insertion du terme.

Rappel : on appelle affectation parallèle une affectation du type `a,b = c,d`.

Exemple 1 :
Voici les étapes de l'exécution du tri par insertion sur la liste `lst = [9,6,1,4,8]`. La liste est représentée au début et à la fin de chaque itération.

$i=1$	**9**	**6**	**1**	**4**	**8**	$\rightarrow$	**6**	**9**	**1**	**4**	**8**
$i=2$	**6**	**9**	**1**	**4**	**8**	$\rightarrow$	**1**	**6**	**9**	**4**	**8**
$i=3$	**1**	**6**	**9**	**4**	**8**	$\rightarrow$	**1**	**4**	**6**	**9**	**8**
$i=4$	**1**	**4**	**6**	**9**	**8**	$\rightarrow$	**1**	**4**	**6**	**8**	**9**

À la fin de la 4ème étape, le dernier terme a été inséré à sa place parmi les précédents donc la liste est triée et l'algorithme prend fin.

EXERCICE 1 : tri par insertion

1. Écrire une fonction `insertion` qui prend en entrée un entier `i` et une liste `lst` dont les `i` premiers éléments sont rangés dans l'ordre croissant et qui insère le terme d'indice `i` à sa place parmi les `i` précédents en utilisant la première stratégie. Cette fonction ne renvoie rien et se code avec une boucle *while* ou avec une boucle *for* et la commande *break* pour sortir de la boucle dès que l'on a trouvé la place du terme à insérer.
   Tester `insertion(4,lst)` avec la liste `lst=[1,4,5,8,2,3]`.
2. Écrire une fonction `tri_insertion` qui prend en entrée une liste `lst` et qui renvoie la liste triée dans l'ordre croissant par l'algorithme de tri par insertion. Cette fonction appellera la fonction *insertion* dans une boucle *for*.
   Tester la fonction `tri_insertion` avec la liste `[5,1,4,8,2,3]`.
3. On cherche à améliorer l'algorithme précédent. Écrire une fonction `insertionD(i,lst)` qui effectue la même tâche que `insertion(i,lst)` en utilisant la deuxième stratégie.
   Tester `insertionD(4,lst)` avec la liste `lst=[1,4,5,8,2,3]`.
4. Écrire une fonction `tri_insertionD(lst)` qui effectue la même tâche que `tri_insertion(lst)` mais qui appelle `insertionD(i,lst)` à la place de `insertion(i,lst)`.
   Tester la fonction `tri_insertionD` avec la liste `[5 1 4 8 2 3]`.
5. (Facultatif) Écrire une fonction `tri_insertion_bis` de paramètre `lst` qui effectue le même travail que `tri_insertion` sans appeler la fonction `insertion`.
   Tester la fonction `tri_insertion_bis` avec la liste `[5,1,4,8,2,3]`.
6. (Facultatif) Écrire une fonction `tri_insertion_ter` de paramètre `lst` qui effectue le même travail que `tri_insertion` sans appeler la fonction `insertion` ni les méthodes *insert* et *pop*. On pourra procéder par une série d'échanges de termes consécutifs par affectations parallèles.
   Tester la fonction `tri_insertion_ter` avec la liste `[5,1,4,8,2,3]`.

Ce tri est celui du joueur de carte qui insère les cartes distribuées les unes après les autres dans un jeu partiellement trié.

2) - Tri par sélection (Selection sort)

Le principe de cet algorithme est de chercher le minimum et de le mettre en première position puis de chercher le second minimum et de le mettre en deuxième position et ainsi de suite. Cette façon de faire est, pour la plupart d'entre nous, la méthode la plus intuitive pour trier.

Dans cet algorithme, on parcourt la liste à trier de l'indice 0 à l'indice $n-2$. À l'étape i on compare le terme d'indice i (en gris foncé dans l'exemple 2) avec le minimum des termes suivants (en gris clair). Si le terme d'indice i est plus grand que ce minimum alors on l'échange avec le premier terme égal à ce minimum sinon on ne fait rien. On exécute la même routine avec le terme d'indice $i+1$ et ainsi de suite.

Exemple 2 :
Voici les étapes de l'exécution du tri par sélection sur la liste `lst = [9,6,1,4,8,10]`. La liste est représentée au début et à la fin de chaque itération.

$i=0$	**9**	**6**	**1**	**4**	**8**	**10**	$\rightarrow$	**1**	**6**	**9**	**4**	**8**	**10**
$i=1$	**1**	**6**	**9**	**4**	**8**	**10**	$\rightarrow$	**1**	**4**	**9**	**6**	**8**	**10**
$i=2$	**1**	**4**	**9**	**6**	**8**	**10**	$\rightarrow$	**1**	**4**	**6**	**9**	**8**	**10**
$i=3$	**1**	**4**	**6**	**9**	**8**	**10**	$\rightarrow$	**1**	**4**	**6**	**8**	**9**	**10**
$i=4$	**1**	**4**	**6**	**8**	**9**	**10**	$\rightarrow$	**1**	**4**	**6**	**8**	**9**	**10**

À la $5^{\text{ème}}$ étape, contrairement aux étapes précédentes, il n'y a pas eu de permutation car $9 \leqslant 10$. La liste est triée et l'algorithme prend fin.

EXERCICE 2 : tri par sélection

1. Rappeler comment on échange les termes d'indices i et j ($i \neq j$) dans la liste *lst* par affectation parallèle.
2. Écrire une fonction `minimo` de paramètres i et `lst` qui renvoie le premier indice du minimum des termes situés au-delà du rang $i-1$. Tester `minimo(3,[5,1,0,8,2,3])` qui doit renvoyer 4.
3. Écrire une fonction `tri_selection(lst)` qui renvoie la liste *lst* triée par l'algorithme de tri par sélection. Cette fonction appellera dans son bloc la fonction *minimo* et procèdera par échanges de termes.
   Tester la fonction `tri_selection` avec la liste `lst=[5 1 4 8 2 3]`.
4. (Facultatif) Écrire une fonction une fonction `tri_selection_bis` de paramètre `lst` qui effectue le même travail que `tri_selection` sans appeler d'autre fonction. Tester la fonction `tri_selection_bis` avec la liste `lst=[5,1,4,8,2,3]`.

3) - Comptage des effectifs d'une série d'entiers et application au tri

Pour une série de nombres entiers on s'intéresse aux effectifs (ou nombre d'occurrences) de toutes les valeurs comprises entre *vmin* et *vmax*.

Exemple 3 :
On considère la série `lst=[7,2,3,4,1,9,6,3,4,3,1,3,7,9,7]` dont les valeurs sont comprises entre $vmin = 1$ et $vmax = 9$. La liste des effectifs pour les entiers de 1 à 9 est `[2,1,4,2,0,1,3,0,2]`. En statistique cette liste est également appelée tableau de fréquences absolues de *lst*.

EXERCICE 3 : tableau de fréquences et tri par comptage (counting sort)

1. Écrire une fonction `tabFreq` d'arguments `lst,vmin,vmax` qui renvoie le tableau de fréquences absolues de la liste d'entiers `lst` pour les valeurs comprises entre les entiers `vmin` et `vmax`.
   Tester `tabFreq([3,1,1,3,1],1,4)` et `tabFreq([3,1,1,3,1],1,2)`.
2. Écrire une fonction `minimaxi` de paramètre `lst` qui renvoie le minimum et le maximum de `lst` sans utiliser les fonctions *min* et *max*. Tester `minimaxi([3,1,2,1,3,1])`.
3. Écrire une fonction `tri_comptage` de paramètre `lst` et qui calcule le tableau de fréquence de la liste d'entiers `lst` entre le minimum et le maximum de `lst` avant de renvoyer la liste triée obtenue en répétant chaque entier autant de fois qu'il apparait dans `lst`. Tester `tri_comptage([3,1,1,2,3,1])`.
4. Si on compare aux tris définis précédemment (tri par insertion, tri par insertion dichotomique, tri par sélection) on remarque que le *tri comptage* a un code relativement simple et, on verra plus loin, qu'il est le plus rapide. Malgré ces avantages, quel est le principal défaut du tri comptage ?

Les tris par insertion, par insertion dichotomique et par sélection procèdent par comparaisons de termes, ils sont donc qualifiés de tris par comparaison. Le tri comptage ne procède pas par comparaisons mais par comptage, il ne fait pas partie des tris par comparaison.

4) - Application du tri par sélection à la détermination de la médiane et des quartiles

Contrairement au tri par insertion, le tri par sélection met les i plus petits éléments de la liste à leur place définitive après la $i^{\text{ème}}$ étape.
Soit n le nombre de termes de *lst*. Si n est impair alors la médiane de *lst* est le terme situé au milieu de la liste triée. Si n est pair alors la médiane de *lst* est la moyenne du dernier terme de la demi-liste triée inférieure et du premier terme de la demi-liste triée supérieure.
Lorsque n est impair on définit la demi-liste inférieure (resp. supérieure) de *lst* comme la demi-liste inférieure (resp. supérieure) de *lst* privée de sa médiane.
On adopte les définitions simplifiées suivantes :

- Le premier quartile de *lst* est la médiane de sa demi-liste inférieure.
- Le troisième quartile de *lst* est la médiane de sa demi-liste supérieure.

EXERCICE 4 : Médiane et quartiles

1. Écrire une fonction `mediane` d'argument `lst` qui renvoie la médiane de la liste de nombre `lst` après avoir trié la moitié inférieure de cette liste par sélection. On commencera le programme par une discussion sur la parité de n.
2. Écrire une fonction `quartile` d'arguments `i,lst` qui renvoie le $i^{\text{ème}}$ quartile de la liste de nombre `lst`. On pourra utiliser la fonction *mediane* pour `lst` ou pour la demi-liste inférieure ou encore pour la demi-liste supérieure. Les demi-listes supérieure et inférieure pourront être obtenues avec une boucle et la méthode *append* ou bien par extraction (*slicing*). Pour les quartiles 1 et 3 on pourra discuter sur la parité de n.

5) - Comparaison des temps d'exécution des différents algorithmes de tri

Exemple 3 :
Importer le module time :
`import time as ti`
Exécuter la commande `ti.time()` à quelques secondes d'intervalle. Que remarque-t-on ? Comment peut-on utiliser la fonction *time* pour mesurer le temps d'exécution d'un programme ?

EXERCICE 5 : Temps d'exécution de `tri_insertion`, `tri_insertionD`, `tri_selection`, `tri_comptage`

1. Écrire une fonction `listeAlea(n=10000)` qui renvoie la liste des n premiers entiers naturels non nuls mélangés de façon aléatoire. On pourra utiliser la commande `np.arange(1,n+1)` qui renvoie le tableau *numpy* des n premiers entiers naturels non nuls, la fonction *list* qui convertit en liste et enfin la fonction *rd.shuffle* qui mélange aléatoirement une liste. On importera les modules nécessaires de la façon suivante : `import time as ti`, `import numpy as np` et `import random as rd`.
2. Écrire une fonction `comparaison(n=10000)` qui renvoie les temps d'exécution des quatre fonctions `tri_insertion`, `tri_insertionD`, `tri_selection`, `tri_comptage` pour une même liste aléatoire générée par la commande `listeAlea()`. Cette liste aléatoire sera copiée (de façon profonde) trois fois grâce à la fonction *list*.
3. Conclure. On modulera cette conclusion par la portée limitée (que l'on rappellera) du *tri comptage*.
   Vu leurs performances, les tris par insertion, par insertion dichotomique et par sélection seront qualifiés de "naïfs".