Exemples de création de listes de nombres aléatoires

- **Création d'une liste de 1000 nombres aléatoires entre 1 et 1000 (pouvant se répéter)**

```
from random import randint
lst = [0]*1000
for i in range(1000):
    lst[i] = randint(1, 1000)
```

```
lst = []
for i in range(1000):
    lst.append(randint(1, 1000))
```

```
from numpy.random import randint
lst = [randint(1, 1001) for i in range(100)]
```

Remarque : random.randint(a, b) renvoie un nombre de [a, b],
numpy.random.randint(a, b) renvoie un nombre de [a, b[.

- **Création d'une liste de 100 nombres aléatoires entre 1 et 1000 sans répétition**

```
from random import sample
lst0 = list(range(1, 1001))
lst = sample(lst0, 100)
```

```
lst = sample(range(1, 1001), 100)
```

- **Création d'une liste de 100 nombres aléatoires entre 1 et 100 sans répétition**

```
from random import shuffle
lst = list(range(1, 101))
shuffle(lst)
```

```
lst = random.sample(range(1,101),100)
```

Commandes de tri implémentées

lst.sort() trie la liste lst.
sorted(lst) renvoie une copie triée de la liste lst.

I. Tri par sélection

1. Une première version simple mais à éviter

```
def tri_selection(lst):
    nlst = []
    for i in range(len(lst)):
        mini = lst[0]
        for x in lst:
            if x < mini:
                mini = x
        nlst.append(mini)
        lst.remove(mini)
    return nlst
```

a. Tester et comprendre la fonction ci-dessus. Pourquoi cette version n'est pas satisfaisante ?

b. Expliquer le principe de fonctionnement de l'algorithme de tri par sélection.

2. Version itérative en place

La version suivante trie la liste en place.

```
def tri_selection_en_place(lst):
    n = len(lst)
    for i in range(n-1):
        mini = lst[i]
        pos_mini = i
        for j in range(i+1, n):
            if lst[j] < mini:
                mini = lst[j]
                pos_mini = j
        lst[i], lst[pos_mini] = lst[pos_mini], lst[i]
    return lst
```

Pour renvoyer une nouvelle liste sans modifier la liste initiale, il suffit d'effectuer une copie de la liste nlst = lst.copy() en début de fonction puis de travailler sur nlst au lieu de lst.

Rappel : on peut parcourir une liste de deux manières :

```
for x in lst:                    for i in range(len(lst)):
    print(x)                         print(lst[i])
```

La méthode de gauche est la plus simple si on n'a pas besoin de connaitre les indices ou de modifier les termes de la liste.

3. Version récursive

```
def tri_selection_rec(lst):
    if len(lst) == 1:
        return lst
    else:
        indice_mini = 0
        for i in range(len(lst)):
            if lst[i] < lst[indice_mini]:
                indice_mini = i
        return [lst[indice_mini]] + tri_selection_rec(
                        lst[:indice_mini] + lst[indice_mini+1:])
```

II. Tri par insertion

Principe : On considère une liste *lst* de nombres dont les éléments de rangs 0 à *i*-1 sont déjà triés, pour $i \geq 1$. On compare alors l'élément de rang *i*, noté *x* à celui de rang *i*-1. Si $x \geq lst[i\text{-}1]$, alors les éléments de rang 0 à *i* sont déjà triés. Sinon, on permute ces deux éléments, et on compare *x* à l'élément de rang *i*-2, puis *i*-3,… ainsi de suite jusqu'à ce que *x* soit à sa place. Les éléments de rang 0 à *i* sont alors triés.
On applique ce procédé en boucle afin de trier tous les éléments de la liste.

Illustration en vidéo : https://www.youtube.com/watch?v=ROalU379l3U

Travail à effectuer : Programmer une fonction de tri par insertion applicable à une liste de nombres.

III. Tri rapide

Principe : On considère une liste *lst* de nombres. Si elle contient un seul élément, elle est déjà triée. Sinon, on choisit un élément *x* (le premier par exemple), et on partage $lst \backslash \{x\}$ en deux listes *linf* et *lsup*, *linf* contenant les éléments inférieurs ou égaux à *x*, *lsup* ceux supérieurs à *x*. La liste triée s'obtient en recollant les listes *linf*, [*x*] et *lsup*, après avoir trié les listes *linf* et *lsup* suivant le même principe.

Illustration en vidéo : https://www.youtube.com/watch?v=ywWBy6J5gz8

Travail à effectuer : Programmer une fonction de tri rapide applicable à une liste de nombres.

IV. Tri fusion

Principe :

- Fusion : On considère deux listes de nombres triées *lst1* et *lst2*. On les fusionne en une liste *lst* également triée. On crée la liste *lst* élément par élément en ajoutant à chaque fois le plus petit élément encore disponible dans les listes *lst1* et *lst2*. Ainsi, le premier élément de *lst* sera le plus petit entre le premier élément de *lst1* et le premier élément de *lst2*. Si on le prend dans *lst1*, pour obtenir le deuxième élément de *lst*, on compare le deuxième élément de *lst1* avec le premier élément de *lst2* et ainsi de suite.

- Tri : Pour trier une liste *lst*, si elle contient un seul élément, elle est déjà triée, sinon on la divise en deux listes que l'on triera de la même manière avant de les fusionner.

Illustration en vidéo : https://www.youtube.com/watch?v=XaqR3G_NVoo

Travail à effectuer : Programmer une fonction de tri fusion applicable à une liste de nombres.

V. Comparaison

Comparer la rapidité des différents algorithmes de tri :

- noter les résultats dans un tableau
- choisir des listes de taille variable
- tester avec des nombres distincts ou pas
- tester les algorithmes sur des listes déjà triées

On pourra utiliser la fonction time de la bibliothèque time.

VI. Application : recherche de la médiane

Ecrire une fonction renvoyant la médiane d'une liste de nombres.

VII. Complément pour les plus rapides

Les plus rapides qui s'ennuient pourront programmer un tri rapide en place (trie la liste lst sans utiliser d'autres listes) ou un tri fusion itératif.