Question 1. Écrire une fonction `generateur` qui prend en argument un entier naturel n et qui renvoie une liste de n entiers aléatoirement choisis (de manière uniforme) dans l'ensemble $[\![1, n]\!]$.
Bonus : il est possible d'écrire la fonction en deux lignes à l'aide de la définition des listes par compréhension.

1 Exemples de tris dits *rapides*

1.1 Le tri fusion (*mergesort*)

Le **tri fusion** est un algorithme de tri basé sur le paradigme "*diviser pour régner*". Son principe est le suivant :

- on partitionne la liste à trier en deux sous-listes de longueurs qui diffèrent d'au plus une unité ;
- on trie *récursivement* les deux sous-listes ;
- on fusionne les deux sous-listes préalablement triées.

Question 2. Recopier et compléter les deux codes ci-dessous afin d'écrire une version récursive et une version itérative d'une fonction qui prend en argument deux listes **triées** `lst1` et `lst2` et qui renvoie une liste triée composée de tous les éléments des deux listes `lst1` et `lst2`.

```
def fusion(lst1, lst2):
    if lst1 == []:
        return ...
    elif ... == []:
        return ...
    else:
        x1 = lst1[...]
        x2 = ...
        if x1 < x2:
            return [...] + fusion(...,...)
        else:
            return ... + fusion(...,...)
```

```
def fusion2(lst1, lst2):
    i1, i2 = 0, 0
    n1, n2 = len(lst1), len(lst2)
    lst = ...
    while i1 + i2 < n1 + n2:
        if i1 == n1:
            return lst + lst2[...:]
        elif ... == ...:
            return lst + ...
        elif lst1[i1] < ...:
            lst.append(lst1[...])
            i1 += ...
        else:
            lst.append(...)
            i2 += ...
    return ...
```

Question 3. Écrire une fonction récursive `tri_fusion` qui prend en argument une liste de nombres et qui renvoie une version triée de cette liste par l'algorithme du tri fusion.
*On ne cherchera pas à écrire un tri **en place**, i.e. un tri qui modifie la liste passée en argument ; on renverra une version triée de la liste passée en argument.*

Question 4. À l'aide de la fonction `time` du module `time`, calculer le temps pour trier par fusion une liste de 10^3 puis 10^4 nombres (générée aléatoirement via la fonction `generateur` de la question 1).
Comparer les temps d'exécutions selon qu'on utilise les fonctions `fusion` ou `fusion2`. Commenter.
En cas de besoin, on pourra utiliser la fonction `setrecursionlimit` du module `sys` afin d'augmenter le nombre maximal d'appels récursifs autorisés (40000 appels récursifs maximum devraient suffire pour le test ci-dessus).

1.2 Tri rapide (*quicksort*)

Le principe du **tri rapide** est similaire au principe du tri fusion : on sépare la séquence à trier en deux sous-séquences qu'on trie récursivement et qu'on fusionne ensuite.
La différence avec le tri fusion réside **dans la manière dont on partitionne** la séquence. Dans l'algorithme de tri fusion, on partitionne en deux sous-séquences de longueurs *équivalentes*. Dans l'algorithme de tri rapide, on choisit une valeur appelée **pivot** et on partitionne la séquence à trier en deux sous-séquences selon que leurs valeurs sont strictement inférieures ou bien supérieures ou égales au pivot.
Le choix du pivot influe a priori sur l'efficacité de l'algorithme ; pour des raisons de simplicité, on choisira la première valeur de la liste (on pourrait choisir au hasard une valeur de la liste).

Question 5. Écrire une fonction `partition` qui prend en argument une liste `lst` de nombres et qui renvoie un triplet `(pivot, inf, sup)` où `pivot` est le premier élément de `lst`, `inf` (resp. `sup`) est la liste des valeurs de `lst`, privé de son premier élément, strictement inférieures (resp. supérieures ou égales) à `pivot`.

Question 6. Écrire une fonction récursive `tri_rapide` qui prend en argument une liste de nombres et qui renvoie une version triée de cette liste par l'algorithme du tri rapide.
On ne cherchera pas à écrire un tri en place.

Question 7. Calculer le temps pour trier à l'aide de l'algorithme de tri rapide une liste de 10^k nombres (générée aléatoirement via la fonction `generateur` de la question 1) pour $k \in [\![4, 6]\!]$. Comparer avec les résultats de la question 4.

2 Principaux tris naïfs (*rappels de sup*)

2.1 Le tri par sélection

Le principe du **tri par sélection** est de placer à la k-ème itération, pour tout indice k d'une liste ou d'un tableau, le k-ème plus petit élément de la séquence à trier en sélectionnant le minimum des valeurs encore non triées. On commence donc par chercher le minimum (de la séquence à trier) qu'on place en première position. On cherche ensuite le minimum des valeurs restantes, qu'on place en seconde position, etc.

Question 8. Écrire une fonction `tri_selection` qui trie *sur place* par sélection une liste de nombres.
La fonction ne devra rien renvoyer[1]*, elle devra simplement modifier la liste passée en argument. On dit que la fonction agit par* ***effet de bord***[2].

Question 9. Calculer le temps pour trier à l'aide de l'algorithme de tri par sélection une liste de 10^k nombres (générée aléatoirement via la fonction `generateur` de la question 1) pour $k \in [\![2, 4]\!]$. Comparer avec les résultats des questions 4 et 7.

2.2 Le tri par insertion

Le principe de l'algorithme de **tri par insertion** est d'insérer successivement chaque élément parmi la collection d'objets précédemment triés, à la manière d'un joueur de cartes.
En pratique :

- Le premier élément du tableau constitue à lui seul une collection déjà triée.
- Pour tout $k \geqslant 1$, on insère l'élément d'indice k en le comparant avec ceux qui le précèdent (par nécessairement tous, puisqu'ils ont été préalablement triés).

[1] En l'absence de `return`, la fonction renvoie par défaut `None`.
[2] Il s'agit d'une mauvaise traduction de l'expression "*side effect*".

Question 10. Recopier et compléter le code ci-dessous afin d'implémenter une version *sur place* de l'algorithme de par insertion un tableau de nombres (de type `list` ou un `array`).

```
def tri_insertion(tab):
    n = len(tab)
    for i in range(..., ...):
        x = tab[i]
        j = i - 1
        while j >= ... and ... :
            tab[...] = tab[...]
            j = ...
        tab[...] = ...
```

Algorithme de tri par insertion

Question 11. Calculer le temps pour trier à l'aide de l'algorithme de tri par insertion une liste de 10^k nombres (générée aléatoirement via la fonction `generateur` de la question 1) pour $k \in [\![2, 4]\!]$. Comparer avec les résultats des questions 4, 7 et 9.

2.3 Le tri bulle

Question 12. La fonction ci-dessous est une implémentation du **tri bulle**. Décrire le principe de l'algorithme en quelques phrases et justifier pourquoi cette fonction trie bien sur place la liste passée en argument.

```
def tri_bulle(liste):
    n = len(liste)
    for i in range(n):
        for j in range(n-i-1):
            if liste[j] > liste[j+1]:
                liste[j], liste[j+1] = liste[j+1], liste[j]
```

Algorithme de tri bulle

Question 13. On peut remarquer que le code de la question 12 ne détecte pas que la liste est triée avant la fin de l'exécution de l'algorithme du tri bulle. Écrire un nouveau code qui réduit le nombre d'itérations de l'algorithme du tri bulle à l'aide d'une boucle conditionnelle (`while`) et d'une variable *drapeau* qui détecte lorsqu'une permutation a été réalisée.

Question 14. Calculer le temps pour trier à l'aide de l'algorithme de tri par insertion une liste de 10^k nombres (générée aléatoirement via la fonction `generateur` de la question 1) pour $k \in [\![2, 4]\!]$. Comparer avec les résultats des questions 4, 7, 9 et 11.

3 Comparaison des performance des algorithmes

Question 15. On souhaite représenter graphiquement les temps moyens d'exécution de tous les algorithmes implémentés dans ce TP. Pour cela, on décide de générer des listes aléatoires de longueur n où n est un entier variant de 100 à 1500 de 100 en 100. Pour chaque valeur de n, et pour chaque algorithme de tri, on générera 10 listes aléatoires de longueur n et on calculera la moyenne des temps pour trier ces listes. Compléter alors le code ci-après et analyser les résultats.

```
longueurs = list(range(100,...,...))
liste_algos = [tri_selection,tri_insertion,tri_bulle,tri_fusion,tri_rapide]
temps = [[] for k in range(5)]
for n in longueurs:
    N = 10
    for k in range(...):
        t = 0
        for i in range(N):
            liste = ...
            algo = liste_algos[k]
            t0 = ...
            algo(...)
            t1 = ...
            t += ...
        temps[...].append(t/N)
for k in range(...):
    algo = liste_algos[k]
    plt.plot(...,...,label=str(algo))
plt.legend(loc = "best")
plt.show()
```