Tous les modules utilisés dans le TP sont importés via les instructions suivantes :

```
import random as rd
import matplotlib.pyplot as plt
import time as t
import sys
```

Question 1.

```
def generateur(n):
    return [rd.randint(1,n) for k in range(n)]
```

1 Exemples de tris dits *rapides*

1.1 Le tri fusion (*mergesort*)

Question 2. Les cas de base de la fonction récursive ci-dessous correspondent aux cas où l'une des deux listes passées en argument est vide.

```
def fusion(lst1, lst2):
    if lst1 == []:
        return lst2
    elif lst2 == []:
        return lst1
    else:
        x1, x2 = lst1[0], lst2[0]
        if x1 < x2:
            return [x1] + fusion(lst1[1:],lst2)
        else:
            return [x2] + fusion(lst1,lst2[1:])
```

```
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[i2:]
        elif i2 == n2:
            return lst + lst1[i1:]
        elif lst1[i1] < lst2[i2]:
            lst.append(lst1[i1])
            i1 += 1
        else:
            lst.append(lst2[i2])
            i2 += 1
    return lst
```

Question 3.

```
def tri_fusion(lst):
    n = len(lst)
    if n <= 1:
        return lst
    else:
        lst1 = tri_fusion(lst[:n//2])
        lst2 = tri_fusion(lst[n//2:])
        return fusion(lst1,lst2)
```

Question 4.

```
def tri_fusion2(lst):
    n = len(lst)
    if n <= 1:
        return lst
    else:
        lst1 = tri_fusion2(lst[:n//2])
        lst2 = tri_fusion2(lst[n//2:])
        return fusion2(lst1,lst2)
```

```
sys.setrecursionlimit(40000)

for k in [3,4]:
    n = 10**k
    print("n=",n)
    liste = generateur(n)
    t0 = t.time()
    tri_fusion(liste)
    t1 = t.time()
    print(t1 - t0)

for k in [3,4]:
    n = 10**k
    liste = generateur(n)
    t0 = t.time()
    tri_fusion2(liste)
    t1 = t.time()
    print(t1 - t0)
```

1.2 Tri rapide (*quicksort*)

Question 5.

```
def partition(lst):
    pivot = lst[0]
    inf, sup = [], []
    for x in lst[1:]:
        if x < pivot:
            inf.append(x)
        else:
            sup.append(x)
    return pivot,inf,sup
```

Question 6.

```
def tri_rapide(lst):
    if len(lst) <= 1:
        return lst
    else:
        pivot,inf,sup = partition(lst)
        l1 = tri_rapide(inf)
        l2 = tri_rapide(sup)
        return l1 + [pivot] + l2
```

Question 7.

```
for k in [3,4,5,6]:
    n = 10**k
    print("n=",n)
    liste = generateur(n)
    t0 = t.time()
    tri_rapide(liste)
    t1 = t.time()
    print(t1 - t0)
```

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, elle devra simplement modifier la liste passée en argument. On dit que la fonction agit par ***effet de bord****.*

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).

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()
```