Tris_2018

July 21, 2018

1 Algorithmes de tri

1.1 Préambule

Quelques liens pour découvrir et comparer des algorithmes de tris :

- Article de Wikipedia
- Article du site Interstices
- Comparaison des algorithmes de tris

Au fait ce texte saisi dans une cellule du notebook d'IPython utilise le langage de balises Markdown, surcouche d'HTML.

1.2 Imports des modules

```
In [11]: import numpy as np
         import matplotlib.pyplot as plt
         import time
         from random import randint

In [12]: %matplotlib inline
```

1.3 Tests de correction

```
In [13]: #Pour tester les fonctions de tri
         #un tableau contenant 3 tableaux d'entiers aléatoires de tailles 10, 100, 1000
         BENCH2 = [[randint(1, 2*10**i) for _ in range(10**i)] for i in range(1,4)]

         #idem mais avec 3 tableaux d'entiers aléatoires de tailles impaires
         BENCH1 = [[randint(1, 2*10**i) for _ in range(10**i + 1)] for i in range(1,4)]

         def bontri(t):
             '''Détermine si un tableau est trié dans l'ordre croissant'''
             for k in range(len(t)-1):
                 if t[k] > t[k+1]:
                     return False
             return True
```

```
def procedure_to_fonction(f):
    '''Remplace la fonction f qui est une procedure ne retournant rien par une
    fonction fbis qui exécute f sur ses arguments puis retourne ses arguments.
    Nécessaire pour composer une fonction de tri sur place avec bontri'''

    def fbis(*args):
        f(*args)
        return args

    return fbis

def test_tri(tri, BENCH):
    #copie profonde de BENCH qui est un tableau de tableaux
    COPIE = [t[:] for t in BENCH]
    return [bontri(procedure_to_fonction(tri)(t)) for t in COPIE]
```

1.3.1 Chronomètre

```
In [14]: def timetest(fonction):
             """exécute la fonction et affiche son temps d'exécution """

             def fonction_modifiee(*args,**kargs):
                 debut = time.perf_counter()
                 fonction(*args,**kargs)
                 return time.perf_counter() - debut

             return fonction_modifiee
```

1.4 Tri par sélection

1.4.1 Implémentations du tri par sélection

```
In [15]: def index_mini(t, debut):
             if t == []:
                 return None
             imini = debut
             for j in range(debut + 1, len(t)):
                 if t[j] < t[imini]:
                     imini = j
             return imini

         def tri_selection(t):
             '''Tri par sélection avec sélection du minimum'''
             n = len(t)
             for debut in range(n - 1):
                 imini = index_mini(t, debut)
                 t[debut], t[imini] = t[imini], t[debut]
```

```
def index_maxi(t, fin):
    if t == []:
        return None
    imaxi = fin
    for j in range(0, fin):
        if t[j] > t[imaxi]:
            imaxi = j
    return imaxi

def tri_selection2(t):
    '''Tri par sélection avec sélection du maximum'''
    n = len(t)
    for fin in range(n - 1, 0, -1):
        imaxi = index_maxi(t, fin)
        t[fin], t[imaxi] = t[imaxi], t[fin]
```

1.4.2 Test de correction du tri par sélection

```
In [16]: for tri in [tri_selection, tri_selection2]:
             print(test_tri(tri, BENCH1))
             print(test_tri(tri, BENCH2))
```

```
[True, True, True]
[True, True, True]
[True, True, True]
[True, True, True]
```

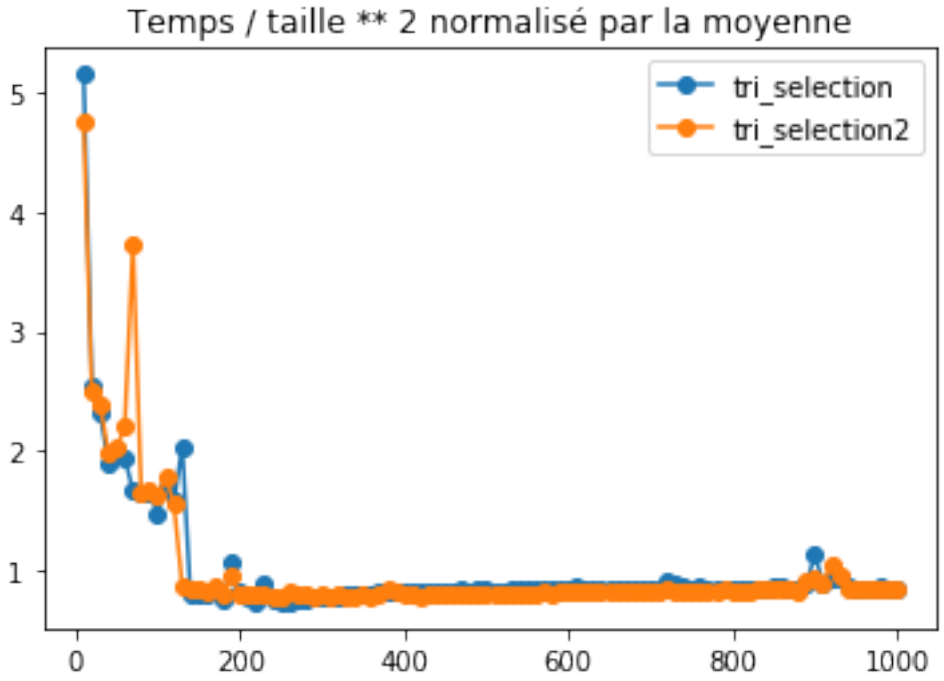
1.4.3 Complexité du tri par sélection

Complexité du tri par sélection pour une liste de n entiers : chaque boucle interne de recherche du maximum dans la partie de $n - i$ valeurs non encore triées, effectue $n - i$ comparaisons ; donc l'algorithme nécessite $(n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2}$ comparaisons. Il s'agit donc d'une **complexité quadratique** que la liste initiale soit déjà triée ou non (comme on peut le constater sur l'exemple ci-dessus).

- Article de Wikipedia
- Le site sorting algorithms

```
In [46]: liste_tris = [tri_selection, tri_selection2]
         liste_taille = list(range(10, 1001, 10))
         liste_rapport_carre = np.array([[0 for _ in range(len(liste_taille))] for _ in range(le
         for i, taille in enumerate(liste_taille):
             liste = [randint(0, taille) for _ in range(taille)]
             for j, tri in enumerate(liste_tris):
                 liste_rapport_carre[j][i] = timetest(tri)(liste[:]) / (liste_taille[i]) ** 2
         for k, rapport_carre in enumerate(liste_rapport_carre):
             plt.plot(liste_taille, rapport_carre / rapport_carre.mean(), label=liste_tris[k].__
         plt.title(r"Temps / taille ** 2 normalisé par la moyenne")
```

```
plt.legend()
plt.savefig('complexite-quadratique-tri_selection.png')
```



1.5 Tri par bulles

L'algorithme du tri par bulles consiste à trier un tableau en ne s'autorisant qu'à échanger deux éléments consécutifs de ce tableau. On peut démontrer que l'algorithme suivant trie n'importe quel tableau :

- chercher deux éléments consécutifs rangés dans le désordre ;
- si deux tels éléments existent, les échanger et recommencer ;
- sinon arrêter.

Descriptions du tri par bulles :

- Article de Wikipedia
- Le site sorting algorithms

1.5.1 Implémentations du tri par bulles

```
In [17]: def tri_bulles(t):
             n = len(t)
             for bulle in range(n - 1):
                 for k in range(0, n - 1 - bulle):
                     if t[k + 1] < t[k]:
```

```
                    t[k + 1], t[k] = t[k], t[k + 1]


        def tri_bulles2(t):
            n = len(t)
            continuer = True
            bulle = 0
            while continuer:
                continuer = False
                for k in range(0, n - 1 - bulle):
                    if t[k + 1] < t[k]:
                        t[k + 1], t[k] = t[k], t[k + 1]
                        continuer = True
                bulle += 1
```

1.5.2 Test de correction du tri par bulles

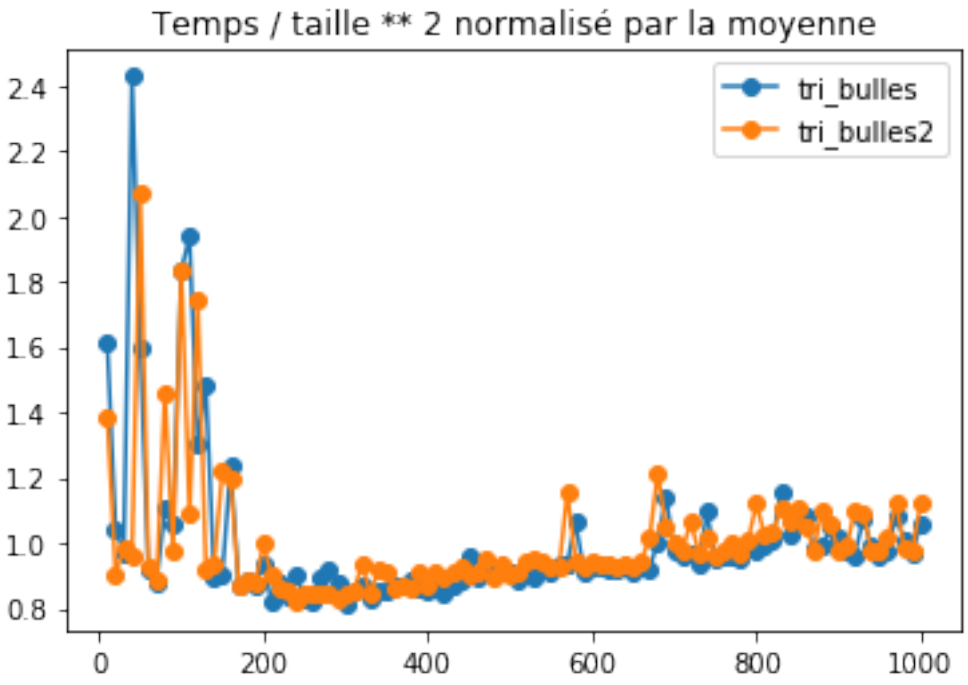
```
In [18]: for tri in [tri_bulles, tri_bulles2]:
             print(test_tri(tri, BENCH1))
             print(test_tri(tri, BENCH2))
```

```
[True, True, True]
[True, True, True]
[True, True, True]
[True, True, True]
```

1.5.3 Complexité du tri par bulles

1. Tri par bulles de la liste [72, 39, 29, 59] :
   - Etape 1 [39,72,29,59]
   - Etape 2 [39,29,72,59]
   - Etape 3 [39,29,59,72]
   - Etape 4 [29,39,59,72]
2. Implémentation en Python du Tri par bulles, coir ci-dessous plusieurs versions.
3. Selon l'implémentation `tri_bulles` ou `tri_bulles2` (voir ci-dessus) la complexité du tri par bulles est **quadratique** (comme pour le tri par sélection) ou **linéaire** (comme pour le tri par insertion) sur *les listes déjà triées*.
4. Si on applique le tri par bulles à une liste triée dans l'ordre décroissant comme $[n, n-1, n-2, .., 1]$, quelle que soit l'implémentation, on a besoin de $i-1$ permutations (comparaison + échange) pour i allant de n à 1 soit un total de $(n-1)+(n-2)+\cdots+1 = \frac{n(n-1)}{2}$ permutations. La complexité du tri par bulles est donc **quadratique** sur les listes triées dans l'ordre inverse.

```
In [45]: liste_tris = [tri_bulles, tri_bulles2]
         liste_taille = list(range(10, 1001, 10))
         liste_rapport_carre = np.array([[0 for _ in range(len(liste_taille))] for _ in range(le
         for i, taille in enumerate(liste_taille):
             liste = [randint(0, taille) for _ in range(taille)]
             for j, tri in enumerate(liste_tris):
                 liste_rapport_carre[j][i] = timetest(tri)(liste[:]) / (liste_taille[i]) ** 2
         for k, rapport_carre in enumerate(liste_rapport_carre):
             plt.plot(liste_taille, rapport_carre / rapport_carre.mean(), label=liste_tris[k].__
         plt.title(r"Temps / taille ** 2 normalisé par la moyenne")
         plt.legend()
         plt.savefig('complexite-quadratique-tri_bulles.png')
```



1.6 Tri par insertion

1. Descriptions du tri par insertion :

   - Article de Wikipedia
   - Le site sorting algorithms

2. Implémentation du tri par insertion en Python (voir ci-dessous)

3. Contrairement au tri par sélection, si l'on interrompt l'exécution de l'algorithme du tri par sélection après k étapes après k étapes, la sous-liste des k premiers éléments déjà triés n'est pas celle des k plus petits élements de la liste triée finale.

1.6.1 Implémentations du tri par insertion

```
In [19]: def insertion(t, k):
             """A partir de l'indice k, insère l'élément t[k] de t
             à sa place dans t[:k] triée dans l'ordre croissant"""
             i = k - 1
             element = t[k]
             while i >= 0 and t[i] > element:
                 t[i + 1] = t[i]
                 i = i - 1
             t[i] = element

         def tri_insertion(t):
             n = len(t)
             for k in range(1, n):
                 insertion(t, k)
```

1.6.2 Test de correction du tri par insertion

```
In [20]: for tri in [tri_insertion]:
             print(test_tri(tri, BENCH1))
             print(test_tri(tri, BENCH2))
```

```
[True, True, True]
[True, True, True]
```

1.6.3 Complexité du tri par insertion

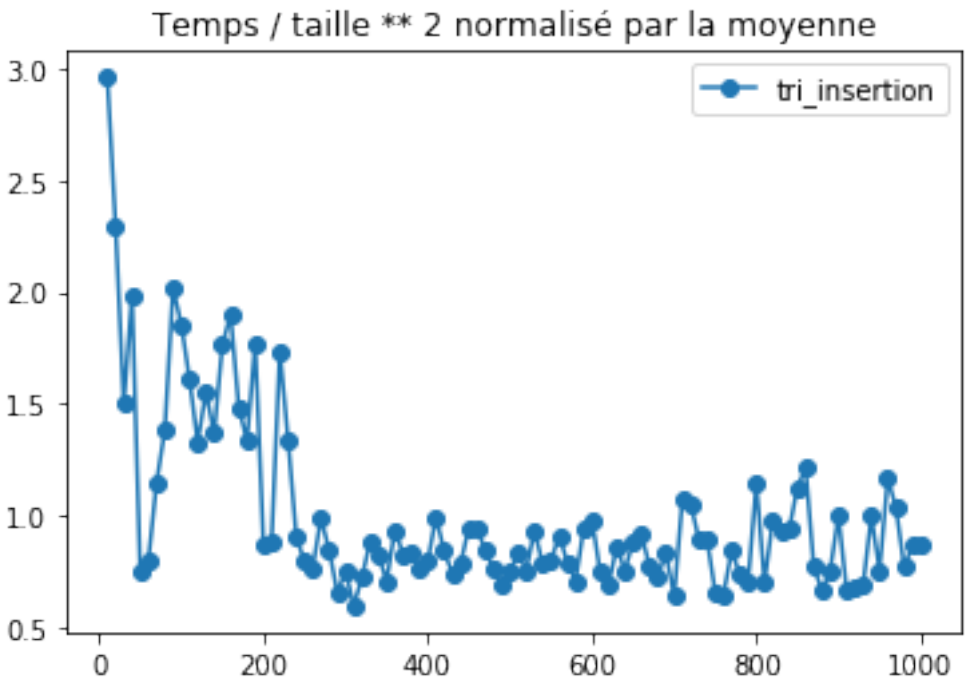
Analyse de complexité du tri par insertion :

- Dans **le meilleur des cas** le tableau (ou liste) est déjà trié, on effectue alors une seule comparaison à chaque insertion soit $n-1$ comparaisons au total
- **En moyenne** (liste d'entiers aléatoires) le nombre de comparaisons est de l'ordre de $n-1+\frac{n(n-1)}{4}$
- **Dans le pire des cas** le tableau est trié dans l'ordre inverse on effectue i comparaisons lors de l'insertion d'indice i soit $1+2+\cdots+(n-2)+(n-1)+\cdots+1=\frac{n(n-1)}{2}$ comparaisons.

La complexité du tri par insertion est donc **quadratique** dans le cas moyen et le pire des cas et **linéaire** dans le meilleur des cas (liste déjà triée). Cette bonne propriété que ne possèdent pas les tris par sélection ou par bulles, font du tri par insertion un tri efficace lorsqu'il y a peu de comparaisons ou que la liste est presque déjà triée.

```
In [44]: liste_tris = [tri_insertion]
         liste_taille = list(range(10, 1001, 10))
         liste_rapport_carre = np.array([[0 for _ in range(len(liste_taille))] for _ in range(le
         for i, taille in enumerate(liste_taille):
             liste = [randint(0, taille) for _ in range(taille)]
             for j, tri in enumerate(liste_tris):
```

```
        liste_rapport_carre[j][i] = timetest(tri)(liste[:]) / (liste_taille[i]) ** 2
for k, rapport_carre in enumerate(liste_rapport_carre):
    plt.plot(liste_taille, rapport_carre / rapport_carre.mean(), label=liste_tris[k].__
plt.title(r"Temps / taille ** 2 normalisé par la moyenne")
plt.legend()
plt.savefig('complexite-quadratique-tri_insertion.png')
```



1.7 Tri fusion

Descriptions du tri par fusion :

- Article de Wikipedia
- Le site sorting algorithms
- Article du site Interstices

1.7.1 Implémentation du tri fusion

```
In [48]: def fusion(liste, p, q, r):
             tampon = []
             i = p
             j = q
             for k in range(r - p):            #fusion des sous-listes triées
                 if j == r or (j < r and i < q and liste[i] <= liste[j]):
                     tampon.append(liste[i])
                 else:
```

```
                tampon.append(liste[j])
        for k in range(r - p):              #recopie de tampon  dans liste[p:r]
            liste[p + k] = tampon[k]

    def tri_fusion(liste, p, r):
        if p + 1 < r:
            q = (p + r) // 2
            tri_fusion(liste, p, q)
            tri_fusion(liste, q, r)
            fusion(liste, p, q, r)
```

1.7.2 Test de correction du tri fusion

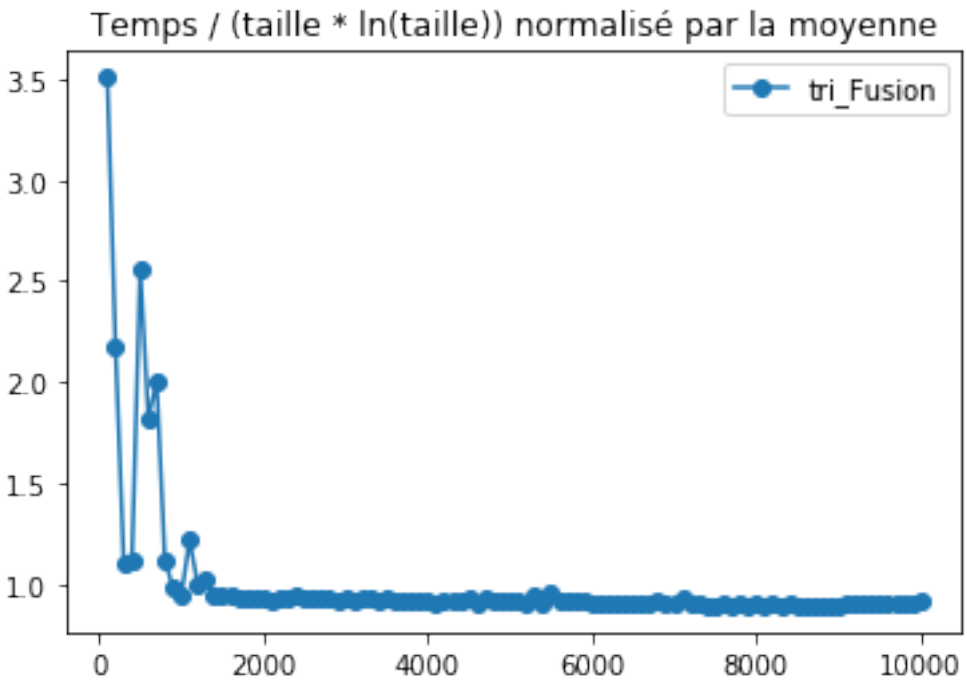
```
In [50]: for tri in [lambda liste : tri_fusion(liste, 0, len(liste))]:
             print(test_tri(tri, BENCH1))
             print(test_tri(tri, BENCH2))

[True, True, True]
[True, True, True]
```

1.7.3 Complexité du tri fusion

La complexité du **tri fusion** sur une liste de taille n est de l'ordre de grandeur de $n\ln(n)$:

- la profondeur de l'arbre d'appels récursifs est de l'ordre de $\frac{\ln(n)}{\ln(2)}$ le logarithme binaire de n
- au niveau k on a 2^k sous-listes de taille $\frac{n}{2^k}$ qu'on fusionne en $2^k \times \frac{1}{2}$ appels à la fonction **fusion** et chaque appel à **fusion** coûte $2 \times \frac{n}{2^k}$ comparaisons ; on a donc $2^k \times \frac{1}{2} \times \frac{n}{2^k} \times 2 = n$ comparaisons par niveau de l'arbre
- le nombre total de comparaisons dans l'arbre est donc bien de l'ordre de grandeur de $\frac{\ln(n)}{\ln(2)} \times n$

```
In [76]: def  tri_Fusion(liste): return tri_fusion(liste, 0, len(liste))
         liste_tris = [tri_Fusion]
         liste_taille = list(range(100, 10001, 100))
         liste_rapport_taille = np.array([[0 for _ in range(len(liste_taille))] for _ in range(l
         for i, taille in enumerate(liste_taille):
             liste = [randint(0, taille) for _ in range(taille)]
             for j, tri in enumerate(liste_tris):
                 liste_rapport_taille[j][i] = timetest(tri)(liste[:]) / (liste_taille[i] * np.lo
         for k, rapport_taille  in enumerate(liste_rapport_taille):
             plt.plot(liste_taille, rapport_taille / rapport_taille.mean(), label=liste_tris[k].
         plt.title(r"Temps / (taille * ln(taille)) normalisé par la moyenne")
         plt.legend()
         plt.savefig('complexite-nln(n)-tri_fusion.png')
```



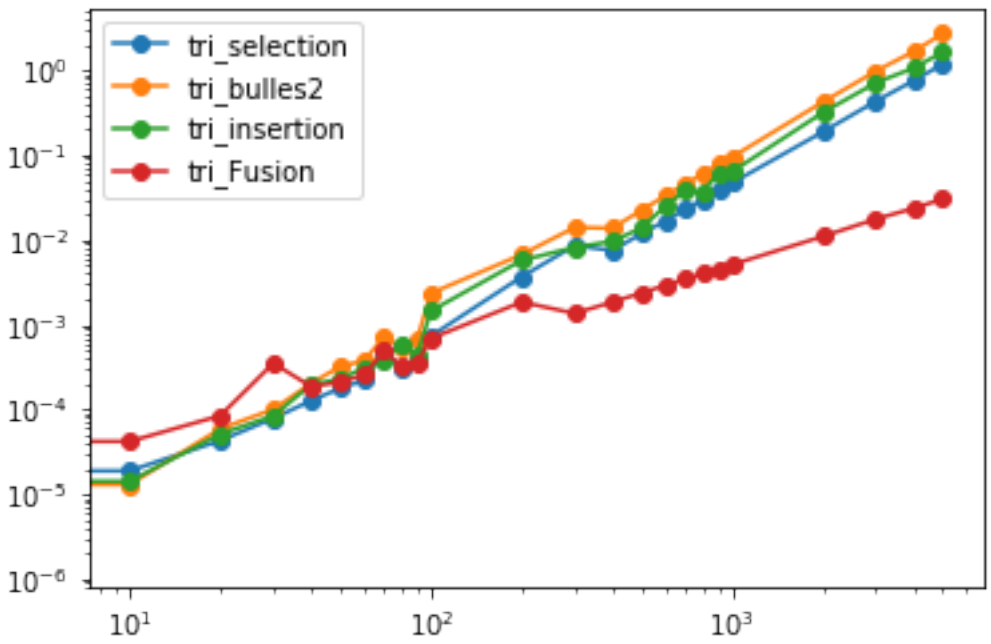
1.8 Comparaison des tris

1.8.1 Les tris en lice

```
In [66]: def  tri_Fusion(liste): return tri_fusion(liste, 0, len(liste))
         liste_tris = [tri_selection, tri_bulles2, tri_insertion, tri_Fusion]
```

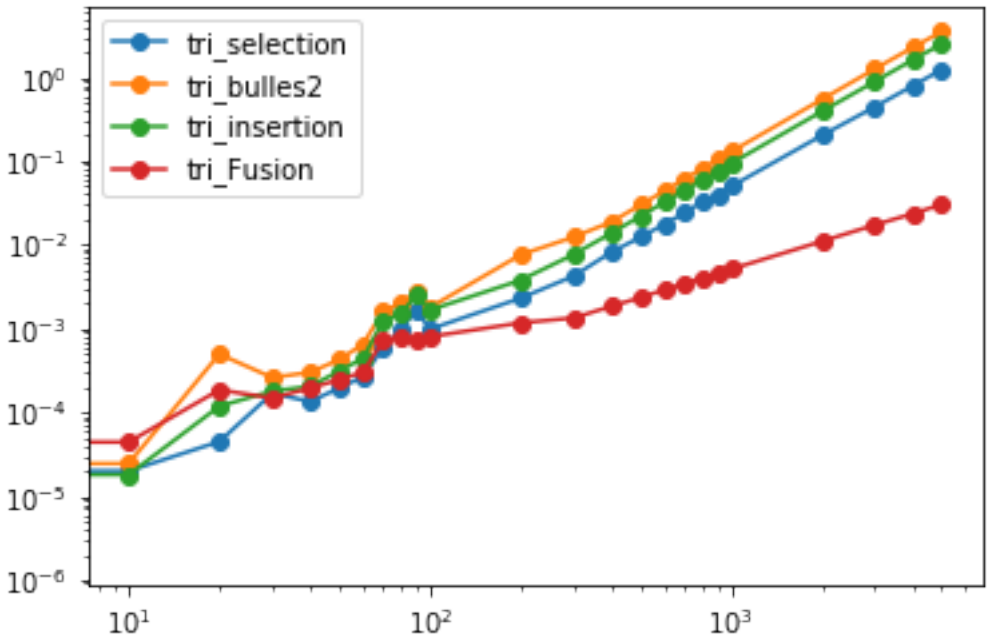
1.8.2 Comparaison des tris sur des listes aléatoires

```
In [71]: liste_taille = list(range(0, 100, 10)) + list(range(100, 1000, 100)) + list(range(1000,
         liste_temps = [[0 for _ in range(len(liste_taille))] for _ in range(len(liste_tris))]
         for i, taille in enumerate(liste_taille):
             liste = [randint(0, taille) for _ in range(taille)]
             for j, tri in enumerate(liste_tris):
                 liste_temps[j][i] = timetest(tri)(liste[:])
         for k, temps in enumerate(liste_temps):
             plt.loglog(liste_taille, temps, label=liste_tris[k].__name__, marker='o')
         plt.legend()
         plt.savefig('test-tris-listes-aleatoires.png')
```



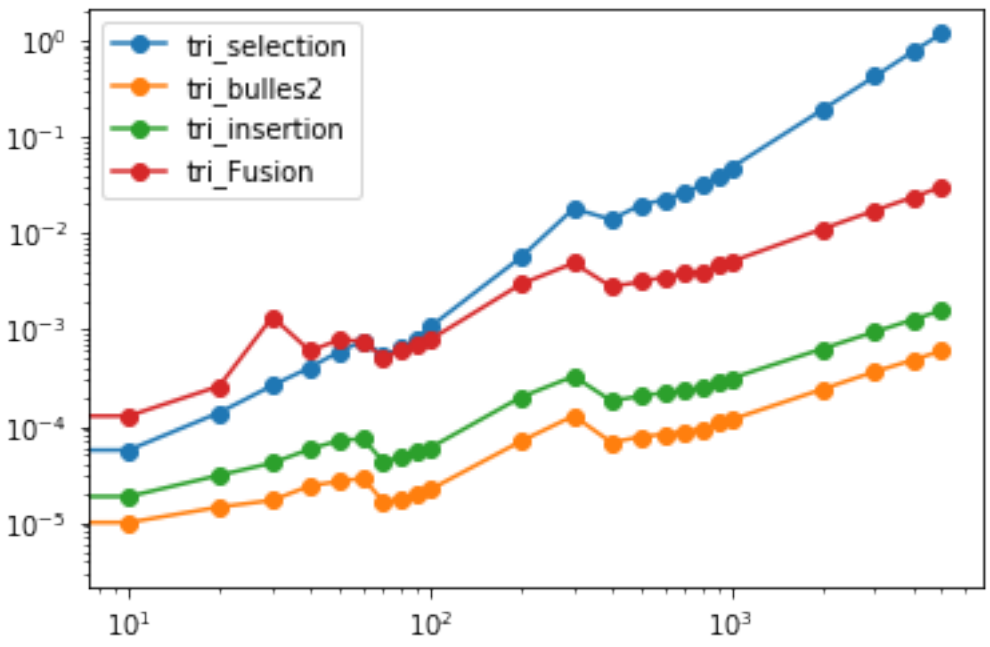
1.8.3 Comparaison des tris sur des listes triées dans l'ordre inverse

```
In [70]: liste_taille = list(range(0, 100, 10)) + list(range(100, 1000, 100)) + list(range(1000,
         liste_temps = [[0 for _ in range(len(liste_taille))] for _ in range(len(liste_tris))]
         for i, taille in enumerate(liste_taille):
             liste = list(range(taille))[::-1]
             for j, tri in enumerate(liste_tris):
                 liste_temps[j][i] = timetest(tri)(liste[:])
         for k, temps in enumerate(liste_temps):
             plt.loglog(liste_taille, temps, label=liste_tris[k].__name__, marker='o')
         plt.legend()
         plt.savefig('test-tris-listes-ordre-inverse.png')
```



1.8.4 Comparaison des tris sur des listes déjà triées

```
In [73]: liste_taille = list(range(0, 100, 10)) + list(range(100, 1000, 100)) + list(range(1000,
         liste_temps = [[0 for _ in range(len(liste_taille))] for _ in range(len(liste_tris))]
         for i, taille in enumerate(liste_taille):
             liste = list(range(taille))
             for j, tri in enumerate(liste_tris):
                 liste_temps[j][i] = timetest(tri)(liste[:])
         for k, temps in enumerate(liste_temps):
             plt.loglog(liste_taille, temps, label=liste_tris[k].__name__, marker='o')
         plt.legend()
         plt.savefig('test-tris-listes-deja-tri.png')
```


tri_selection
tri_bulles2
tri_insertion
tri_Fusion
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{1}
10^{2}
10^{3}