TP noté : Algorithme de tri par ordre

Introduction

Le tri à bulles est l'un des algorithmes de tri les plus lents que nous avons vu. Son principe est de faire remonter les grandes valeurs rapidement à la fin de la liste. Cependant, les petites valeurs qui peuvent se trouver en fin de liste mettent beaucoup de temps à redescendre en début de liste. Le but de ce TP est de développer deux variantes (très proches) du tri à bulles qui pallient à cet inconvénient.

Vous devez importer le fichier `mesFonctionsTri.py`. Vous en trouverez une version propre sur la page du cours dans le dossier `TD_note`. Votre travail de TP doit être stocké dans un fichier nommé par votre prénom et nom sous le format suivant : `triOrdrePrenomNom.py`. Vous devrez à la fin envoyer ce fichier ainsi que les figures que vous aurez sauvegardées par mail à `alia.abbara@ens.fr` et `elie.oriol@phys.ens.fr`.

Pour éviter tout problème dû aux importations, on vous donne le morceau de code suivant que vous devez copier au début de votre fichier :

```
import numpy as np
import matplotlib.pyplot as plt
import random
import time

import os
os.chdir("...")
from mesFonctionsTri import *
```

où à la place des points de suspension dans la commande `os.chdir("...")` vous spécifierez le chemin de votre dossier de travail, où se trouve le fichier `mesFonctionsTri.py`. Pour préciser le chemin, on veillera à l'écrire avec des slashs `"/"` ou des doubles antislashs `"\\"`, mais pas avec des simples antislashs `"\"`. Exemple :

```
        os.chdir("/Users/Moi/Documents/TD_note")
OU      os.chdir("\\Users\\Moi\\Documents\\TD_note")
```

Pour obtenir le chemin du dossier où se trouve votre fichier python, on rappelle la commande `os.getcwd()`

Principe de l'algorithme

L'idée est d'effectuer ce qu'on va appeler des tris à bulles d'ordre k, successivement. Au lieu de reposer sur la comparaison et l'échange d'éléments séparés d'une seule position de la liste à trier, ces tris à bulles reposent sur la comparaison et l'échange d'éléments séparés de k positions : on parcourt la liste, mais au lieu de comparer des éléments aux positions j et $j+1$, on compare des éléments aux positions j et $j+k$, en les échangeant si besoin. On va alors débuter avec une haute valeur de k, effectuer un tri à bulles d'ordre k, et répéter ce procédé en diminuant la valeur de k progressivement jusqu'à $k=1$, ce qui revient à terminer par un tri à bulles classique. L'algorithme proposé est donc le suivant :

```
Algorithme : Tri par ordre
Arguments : Liste L de taille n
Renvoie : rien (modifie la liste en place)

Pour k allant de n-1 a 1
    Effectuer tri a bulles d ordre k
```

Important : on se basera sur la version optimisée du tri à bulles définie au TD4 (Exercice 5 question 3) pour écrire les tris à bulles d'ordre k.

1. Ecrire une fonction `triOrdre0()` prenant en argument une liste L et effectuant le tri par ordre défini ci-dessus.

2. Ecrire une fonction prenant en argument un entier n et renvoyant une liste de longueur n remplie de nombres aléatoires dans l'intervalle $[-1000; 1000]$. Cette fonction sera utile pour toute la suite. L'utiliser pour générer une liste de longueur $n = 5000$ et mesurer le temps d'éxécution de `triOrdre0()` sur cette liste.

Correction

1.
```
def triBullesK(L, k):
    for i in range(len(L)-k, 0, -1):
        fini = True
        for j in range(i):
            if L[j+k] < L[j]:
                L[j+k], L[j] = L[j], L[j+k]
                fini = False
        if fini:
            return


def triOrdre0(L):
    for k in range(len(L)-1, 0, -1):
        triBullesK(L, k)
```

2.
```
def listeAlea(n):
    l=[]
    for i in range(n):
        l.append(random.uniform(-1000,1000))
    return l

L = listeAlea(5000)
t0 = time.time()
triOrdre0(L)
print("Temps d'execution de triOrdre0() :", time.time() - t0)
```

Sur la machine du prof, on obtient un temps de l'ordre de $3.5\,s$.

Première amélioration

On observe que l'algorithme est très lent. Cela est du au fait que diminuer la valeur de k par pas de 1 est loin d'être optimal. On propose de diminuer la valeur de k d'un facteur $f > 1$, de telle sorte qu'après avoir effectué un tri à bulles d'ordre k, on effectue un tri à bulles d'ordre $k' = \lfloor k/f \rfloor$, en diminuant ensuite toujours l'ordre jusqu'à 1. On commence à l'ordre $k = \lfloor n/f \rfloor$ où n est la longueur de la liste à trier.

3. Ecrire une fonction `triOrdre1()` prenant en argument une liste L et un facteur de réduction f et effectuant un tri par ordre dans lequel les ordres sont successivement divisés par f tel que proposé ci-dessus.

4. Générer une liste de longueur $n = 5000$ et mesurer pour $f = 1.2$ le temps d'éxécution de `triOrdre1()` sur cette liste. Comparer à la performance de `triOrdre0()`.

Correction

3.
```
def triOrdre1(L, f):
    k = len(L)
    while k > 1:
        k = int(k/f)
        if k <= 1:
            triBullesOpti(L)
        else:
            triBullesK(L, k)
```

4.
```
L = listeAlea(5000)
t0 = time.time()
triOrdre1(L, 1.2)
print("Temps d'execution de triOrdre1() :", time.time() - t0)
```

Sur la machine du prof, on obtient un temps de l'ordre de $0.9\,s$.

Seconde amélioration

L'algorithme est déjà plus rapide. Il est cependant encore possible de l'améliorer. En effet, lors de l'éxécution des tris à bulles d'ordre $k > 1$, il n'est pas utile de parcourir la liste plusieurs fois. On propose donc de simplifier les tris à bulles effectués en ne leur faisant parcourir chacun la liste qu'une seule fois : pour un tri à bulles d'ordre k, on parcourt la liste une seule fois du début à la fin en comparant les éléments aux positions i et $i + k$ et en les échangeant si besoin, et on s'arrête là.

On voit que dans ce cas, la liste risque de ne pas être totalement triée. Pour répondre à ce problème, on doit garder un tri à bulles classique pour la fin : lorsque $k = 1$, on effectue un tri à bulles classique qui parcourt la liste plusieurs fois.

5. Ecrire une fonction `triOrdre2()` prenant en argument une liste L et un facteur de réduction f et qui implémente la modification ci-dessus.

6. Générer une liste de longueur $n = 5000$ et mesurer pour $f = 1.2$ le temps d'éxécution de `triOrdre2()` sur cette liste. Comparer à la performance de `triOrdre0()` et `triOrdre1()`.

Correction

5.
```
def triOrdre2(L, f):
    k = len(L)
    while k > 1:
        k = int(k/f)
        if k <= 1:
            triBullesOpti(L)
        else:
            for i in range(len(L)-k):
                if L[i] > L[i+k]:
                    L[i], L[i+k] = L[i+k], L[i]
```

6.
```
L = listeAlea(5000)
t0 = time.time()
triOrdre2(L, 1.2)
print("Temps d'execution de triOrdre2() :", time.time() - t0)
```

Sur la machine du prof, on obtient un temps de l'ordre de $0.3\,s$.

Recherche du f optimal

On se propose maintenant de déterminer la valeur optimale de f. On va pour cela tester différentes valeurs de f. On tracera des figures pour chacune des questions suivantes. On demandera de les titrer et d'en nommer les axes.

7. Générer une liste de longueur $n = 10000$ et calculer pour f allant de 1.1 à 2.0 inclus par pas de 0.1 les temps d'éxécution de `triOrdre1()` et `triOrdre2()` sur cette liste. Afficher dans une figure ces temps d'éxécution en fonction de f et la sauvegarder dans le fichier `prenomNom7.png`. On légendera la figure pour distinguer les courbes de `triOrdre1()` et `triOrdre2()`. Que constatez-vous ?

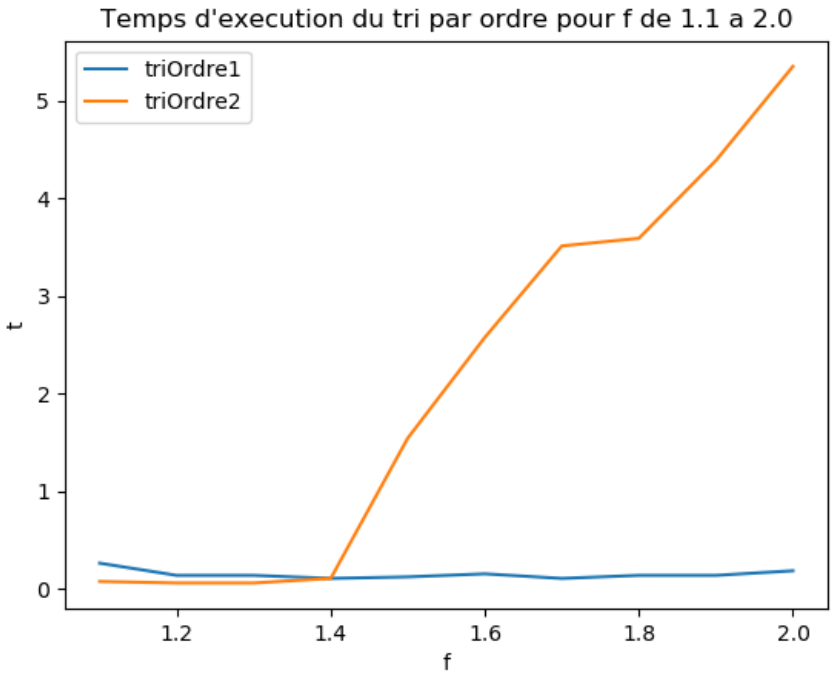
8. On propose de regarder plus précisément l'intervalle où `triOrdre2()` est le plus efficace, c'est-à-dire pour f allant de 1.2 à 1.4 inclus par pas de 0.02. Aussi, effectuer une mesure de temps sur une seule liste n'est pas précis. On veut moyenner les temps d'éxécution obtenus sur un certain nombre de listes. On choisit de prendre ce nombre égal à 10. Afficher dans une nouvelle figure les temps d'éxécution moyennés obtenus pour `triOrdre1()` et `triOrdre2()` en fonction de f et la sauvegarder dans le fichier `prenomNom8.png`. Donnez la différence entre le minimum de temps obtenu pour `triOrdre2()` et celui obtenu pour `triOrdre1()`.

9. Récupérer la valeur de f pour laquelle `triOrdre2()` est le plus rapide. On pourra pour cela utiliser la fonction `argmin(a)` de `numpy` qui prend en argument une liste `a` et renvoie l'index de son minimum. Générer des listes de taille $n = 100$ à $n = 100000$ par pas logarithmiques. On utilisera pour cela la fonction `logspace(start=2, stop=5, num=20, dtype='int')` de `numpy` qui prend comme arguments une valeur de départ `start` ($\log(100) = 2$), une valeur de fin `stop` ($\log(100000) = 5$), un nombre `num` qu'on prendra égal à 20 et un type `dtype` qu'on fixe à `'int'`. Afficher les temps d'éxécution moyennés sur 10 listes en fonction de n pour `triOrdre2()` et pour la fonction `triRapideEnPlace()` de `mesFonctionsTri.py` et la sauvegarder dans le fichier `prenomNom9.png`. En plus de titrer et nommer les axes, on légendera la figure et on mettra l'axe des abscisses et l'axe des ordonnées en échelle logarithmique. En déduire la complexité de l'algorithme.

Correction

7.
```
plt.figure('Question 7')

n = 10000
L = listeAlea(n)
fs = np.arange(1.1, 2.1, .1)
num_fs = len(fs)
tps1 = np.zeros(num_fs)
tps2 = np.zeros(num_fs)
for i in range(num_fs):
    f = fs[i]

    t0 = time.time()
    triOrdre1(L[:], f)
    tps1[i] = time.time()-t0

    t0 = time.time()
    triOrdre2(L[:], f)
    tps2[i] = time.time()-t0

plt.plot(fs, tps1, label='triOrdre1')

plt.plot(fs, tps2, label='triOrdre2')
```

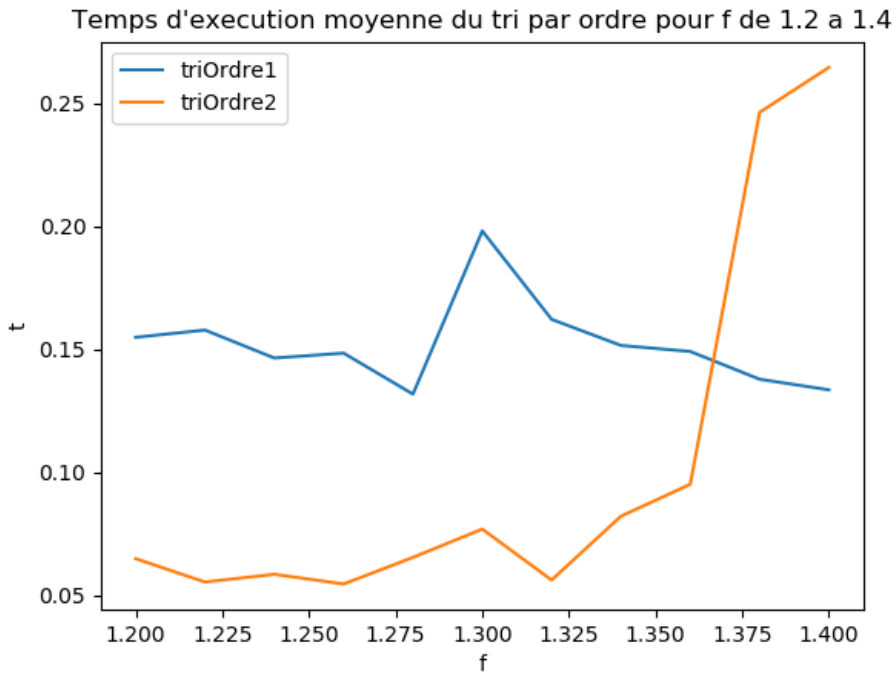
```
plt.title("Temps d'execution du tri par ordre pour f de 1.1 a 2.0")
plt.xlabel('f')
plt.ylabel('t')
plt.legend()
plt.show()
plt.savefig('7.png')
```



8.
```
plt.figure('Question 8')

n = 10000
fs = np.arange(1.2, 1.42, .02)
num_fs = len(fs)
tps1 = np.zeros(num_fs)
tps2 = np.zeros(num_fs)
reps = 10
for rep in range(reps):
    L = listeAlea(n)
    for i in range(num_fs):
        f = fs[i]

        t0 = time.time()
        triOrdre1(L[:], f)
        tps1[i] += time.time()-t0

        t0 = time.time()
        triOrdre2(L[:], f)
        tps2[i] += time.time()-t0

tps1 /= reps
tps2 /= reps

plt.plot(fs, tps1, label='triOrdre1')

plt.plot(fs, tps2, label='triOrdre2')
```

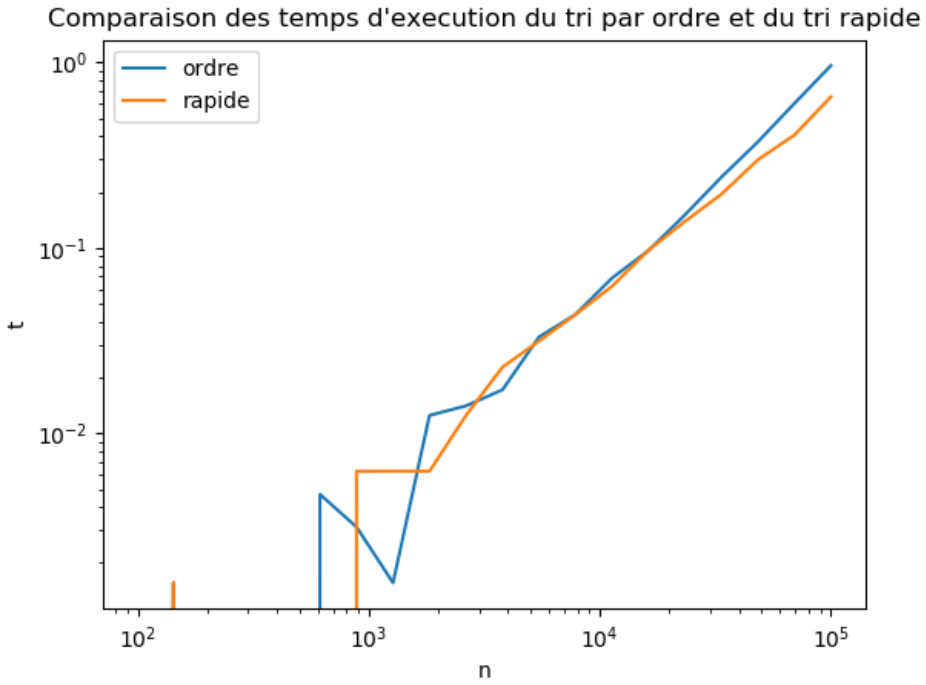
```
plt.title("Temps d'execution moyenne du tri par ordre pour f de 1.2 a 1.4")
plt.xlabel('f')
plt.ylabel('t')
plt.legend()
plt.show()
plt.savefig('8.png')

print("Difference des minima :", min(tps2)-min(tps1))
```

Sur la machine du prof, on obtient une différence des minima de l'ordre de $-0.07\,s$.



9.
```
plt.figure('Question 9')

f = fs[np.argmin(tps2)]
num_tailles = 20
tailles = np.logspace(2, 5, num_tailles, dtype='int')
tps_ord = np.zeros(num_tailles)
tps_rap = np.zeros(num_tailles)
reps = 10
for rep in range(reps):
    for i in range(num_tailles):
        L = listeAlea(tailles[i])

        t0 = time.time()
        triOrdre2(L[:], f)
        tps_ord[i] += time.time()-t0

        t0 = time.time()
        triRapideEnPlace(L[:])
        tps_rap[i] += time.time()-t0

tps_ord /= reps
tps_rap /= reps
```

```
plt.plot(tailles, tps_ord, label='ordre')

plt.plot(tailles, tps_rap, label='rapide')

plt.title("Comparaison des temps d'execution du tri par ordre et du tri rapide"
    )
plt.xlabel('n')
plt.ylabel('t')
plt.xscale('log')
plt.yscale('log')
plt.legend()
plt.show()
plt.savefig('9.png')
```



Une dernière amélioration

Il est possible d'améliorer encore l'algorithme en remplaçant le tri à bulles classique final par un tri par insertion. Le tri par insertion est en effet très efficace sur des listes presque triées, cas dans lequel sa complexité est linéaire. Les tris à bulles d'ordre k effectués dans la première partie de l'algorithme servent alors à obtenir une liste presque triée sur laquelle on éxécute un tri par insertion.

10. Ecrire une fonction `triOrdreInsertion()` sur ce modèle. Effectuer la même étude qu'à la question 8 avec `triOrdreInsertion()` et `triOrdre2()` en sauvegardant une nouvelle figure `prenomNom10.png`.

11. Effectuer la même étude qu'à la question 9 en affichant `triOrdreInsertion(),triOrdre2()` et `triRapide()` et en sauvegardant une nouvelle figure `prenomNom11.png`.

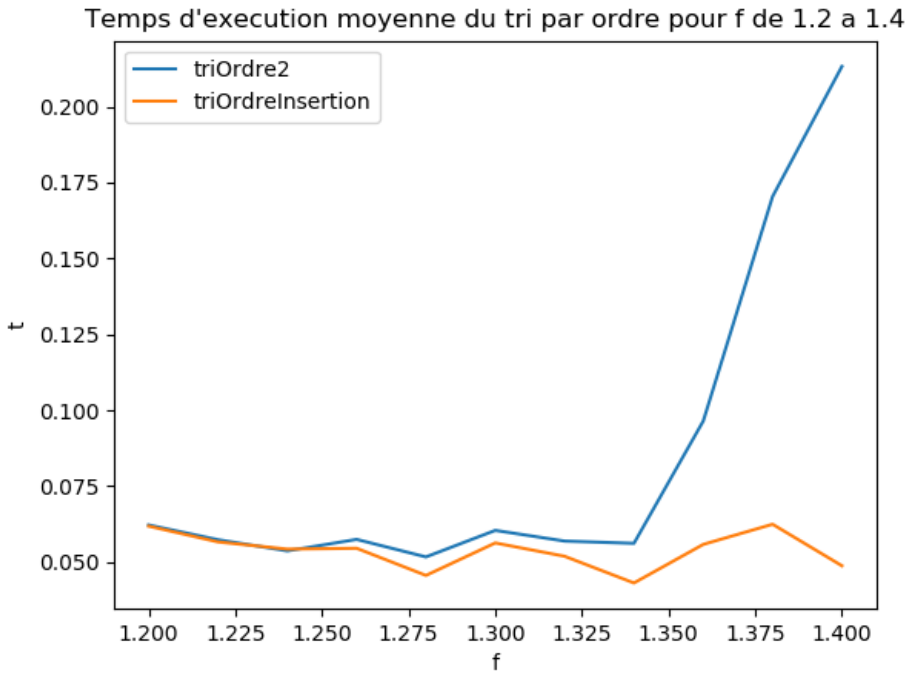
Correction

10.
```
def triOrdreInsertion(L, f):
```

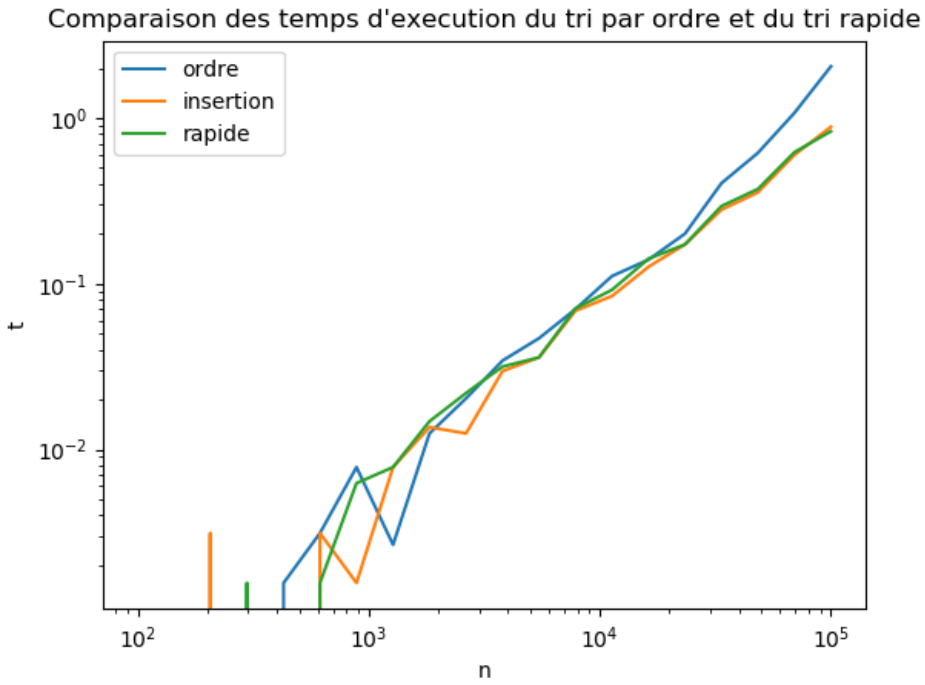
```
    k = len(L)
    while k > 1:
        k = int(k/f)
        if k <= 1:
            triInsertion(L)
        else:
            for i in range(len(L)-k):
                if L[i] > L[i+k]:
                    L[i], L[i+k] = L[i+k], L[i]

plt.figure('Question 10')

n = 10000
fs = np.arange(1.2, 1.42, .02)
num_fs = len(fs)
tps1 = np.zeros(num_fs)
tps2 = np.zeros(num_fs)
reps = 10
for rep in range(reps):
    L = listeAlea(n)
    for i in range(num_fs):
        f = fs[i]

        t0 = time.time()
        triOrdre2(L[:], f)
        tps1[i] += time.time()-t0

        t0 = time.time()
        triOrdreInsertion(L[:], f)
        tps2[i] += time.time()-t0

tps1 /= reps
tps2 /= reps

plt.plot(fs, tps1, label='triOrdre2')

plt.plot(fs, tps2, label='triOrdreInsertion')

plt.title("Temps d'execution moyenne du tri par ordre pour f de 1.2 a 1.4")
plt.xlabel('f')
plt.ylabel('t')
plt.legend()
plt.show()
plt.savefig('10.png')

print("Difference des minima :", min(tps2)-min(tps1))
```

Sur la machine du prof, on obtient une différence des minima de l'ordre de $-0.008\,s$.

11.
```
plt.figure('Question 11')

f1, f2 = fs[np.argmin(tps1)], fs[np.argmin(tps2)]
num_tailles = 20
tailles = np.logspace(2, 5, num_tailles, dtype='int')
tps_ord = np.zeros(num_tailles)
tps_oi = np.zeros(num_tailles)
tps_rap = np.zeros(num_tailles)
reps = 10
for rep in range(reps):
    for i in range(num_tailles):
        L = listeAlea(tailles[i])
```




```
        t0 = time.time()
        triOrdre2(L[:], f1)
        tps_ord[i] += time.time()-t0

        t0 = time.time()
        triOrdreInsertion(L[:], f2)
        tps_oi[i] += time.time()-t0

        t0 = time.time()
        triRapideEnPlace(L[:])
        tps_rap[i] += time.time()-t0

tps_ord /= reps
tps_oi /= reps
tps_rap /= reps

plt.plot(tailles, tps_ord, label='ordre')

plt.plot(tailles, tps_oi, label='insertion')

plt.plot(tailles, tps_rap, label='rapide')

plt.title("Comparaison des temps d'execution du tri par ordre et du tri rapide"
    )
plt.xlabel('n')
plt.ylabel('t')
plt.xscale('log')
plt.yscale('log')
plt.legend()
plt.show()
plt.savefig('11.png')
```



Bonus : Application similaire au tri par insertion

On peut appliquer le même type d'idée développée ici pour le tri à bulles au tri par insertion. De la même manière qu'on a défini des tris à bulles d'ordre k, on peut définir des tris par insertion d'ordre k qui ne comparent non plus des voisins espacés de 1 position mais des éléments espacés de k positions. L'algorithme consiste alors à partir de k égal à la taille de la liste et de le diminuer d'un facteur f donné à chaque étape jusqu'à la valeur de $k = 1$ pour laquelle on finit avec un tri par insertion classique.

12. Appliquer cette idée au tri par insertion en écrivant une nouvelle fonction.

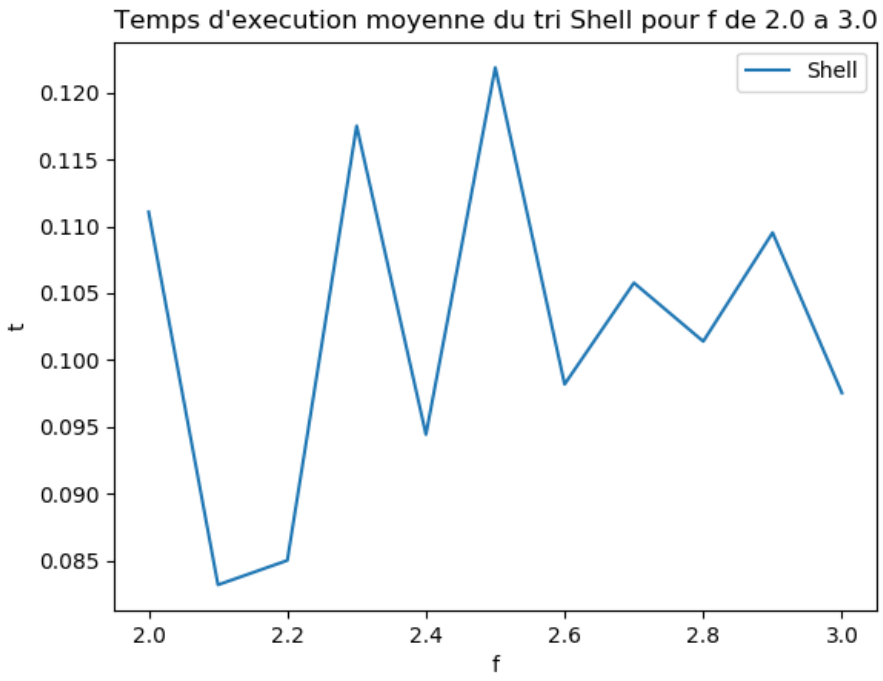
13. Trouver le facteur de réduction f optimal associé à ce nouvel algorithme. On cherchera f entre 2 et 3 par pas de 0.1.

14. Reprendre l'étude de la question 11 avec ce nouveau tri, `triOrdreInsertion()` et `triRapide()`. En déduire la complexité de l'algorithme.

Correction

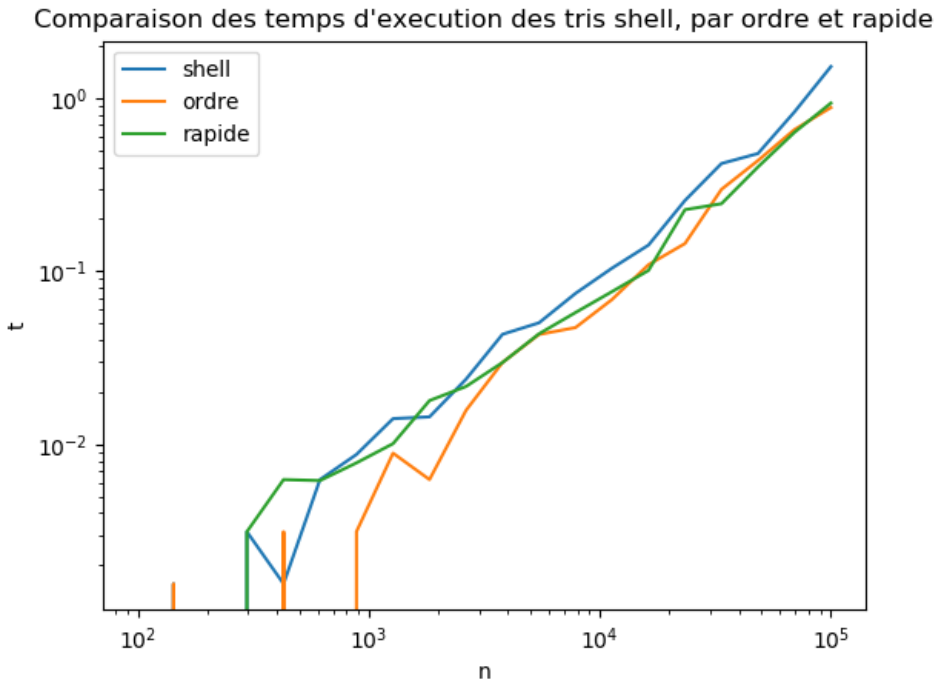
12.
```
def triShell(L, f):
    k = len(L)
    while k > 1:
        k = int(k/f)
        if k <= 1:
            triInsertion(L)
        else:
            for i in range(k, len(L)):
                j = i-k
                while j >= 0 and L[j+k] < L[j]:
                    L[j], L[j+k] = L[j+k], L[j]
                    j -= k
```

13.
```
plt.figure('Question 13')

n = 10000
fs = np.arange(2, 3.1, .1)
num_fs = len(fs)
tps = np.zeros(num_fs)
reps = 10
for rep in range(reps):
    L = listeAlea(n)
    for i in range(num_fs):
        f = fs[i]

        t0 = time.time()
        triShell(L[:], f)
        tps[i] += time.time()-t0

tps /= reps

plt.plot(fs, tps, label='Shell')


plt.title("Temps d'execution moyenne du tri Shell pour f de 2.0 a 3.0")
plt.xlabel('f')
plt.ylabel('t')
plt.legend()
plt.show()
plt.savefig('13.png')
```



14.
```
plt.figure('Question 14')

f3 = fs[np.argmin(tps)]
num_tailles = 20
tailles = np.logspace(2, 5, num_tailles, dtype='int')
tps_she = np.zeros(num_tailles)
tps_oi = np.zeros(num_tailles)
```

```
tps_rap = np.zeros(num_tailles)
reps = 10
for rep in range(reps):
    for i in range(num_tailles):
        L = listeAlea(tailles[i])

        t0 = time.time()
        triShell(L[:], f3)
        tps_she[i] += time.time()-t0

        t0 = time.time()
        triOrdreInsertion(L[:], f2)
        tps_oi[i] += time.time()-t0

        t0 = time.time()
        triRapideEnPlace(L[:])
        tps_rap[i] += time.time()-t0

tps_she /= reps
tps_oi /= reps
tps_rap /= reps

plt.plot(tailles, tps_she, label='shell')

plt.plot(tailles, tps_oi, label='ordre')

plt.plot(tailles, tps_rap, label='rapide')

plt.title("Comparaison des temps d'execution des tris shell, par ordre et
    rapide")
plt.xlabel('n')
plt.ylabel('t')
plt.xscale('log')
plt.yscale('log')
plt.legend()
plt.show()
plt.savefig('14.png')
```


Comparaison des temps d'execution des tris shell, par ordre et rapide
shell
ordre
rapide
10^0
10^{-1}
10^{-2}
t
10^2
10^3
10^4
10^5
n