Tri par fusion corrige

December 13, 2021

1 Principe

Le tri fusion est une méthode de tri récursive.

Pour trier une liste L de taille n :

- elle divise d'abord la liste en deux sous-liste de taille n/2, par exemple L_1 = L[:n//2] et L_2 = L[n//2:]
- elle s'appelle elle même pour trier les deux sous-listes de taille n/2, tri_fusion(L_1) et tri_fusion(L_2)
- elle fusionne les deux sous-listes triées pour obtenir L.

Comme toute fonction récursive, elle doit avoir un critère d'arrêt : ici il s'agit d'une liste de taille n=1, c'est une liste déjà triée. Et on doit s'assurer de la terminaison du programme : ici le critère d'arrêt est atteint pour une taille de liste en argument de 1, et la taille des listes appelées est divisée par 2 à chaque appel récursif donc on atteint bien le critère d'arrêt. En effet on appelle tri fusion pour une liste de taille n, puis n/2, puis n/4, ... , 1.

L'opération de division de la liste à trier en deux est réalisée grace aux opérations sur les indices.

L'opération de fusion consiste à remplir successivement la liste finale avec le plus petit élément entre les deux sous-listes déjà triés. La liste finale se remplie alors du plus petit élément au plus grand. Cette procédure peut se faire car les sous-listes étant déjà triées, on a accès à leur plus petit élément comm étant le premier élément de la sous-liste.

Cette procédure d'appel récursif pour des objets de taille divisé par 2 à chaque appel, est appelée diviser pour régner. On "divise" le problème en triant séparement chaque moitié de liste, on "règne" en fusionnant les deux sous-listes plus facile à trier séparement.

2 Exemple de tri par fusion

Déroulons les opérations sucessives du tri par fusion sur un exemple $L = [14,\ 1,\ 4,\ 3]$:

- on n'est pas au cas d'arrêt donc on divise L -> L_1 = [14, 1] et L_2 = [4, 3]
- on n'est pas au cas d'arrêt donc on divise L_1 -> L_{11} = [14] et L_{12} = [1] et L_2 -> L_{21} = [4] et L_{22} = [3]
- on est au cas d'arrêt les sous-listes L_{11}, L_{12}, L_{21}, et L_{22} sont déjà triées
- on fusionne L_{11} = [14] et L_{12} = [1] comme L_{11} = 14 > L_{12} =1 on a L_1 = [L_{12}, L_{11}] = [1, 14]

- on fusionne L_{21} = [4] et L_{22} = [3] comme L_{21} = 4 > L_{22} = 3 on a L_2 = [L_{22}, L_{21}] = [3, 4]
- on fusionne L_1 = [1, 14] et L_2 = [3, 4] en plusieurs étapes :
  - comme 1 < 3 on rempli L avec 1 donc L = [1]
  - puis on compare [14] et [3, 4]: comme 14 > 3 on rempli L avec 3 donc L = [1, 3]
  - puis on compare [14] et [4]: comme 14 > 4 on rempli L avec 4 donc L = [1, 3, 4]
  - enfin on place 14 donc L = [1, 3, 4, 14]

3 Réalisation

Ecrivons l'algorithme de tri en le décomposant en chaque fonction élémentaire.

La première fonction doit diviser une liste en deux sous-listes de taille moitiés.

La foncion ci-dessous prend en argument une liste L de taille n>1 et renvoie un tuple formé de deux listes de taille n/2

[1]:
```
def diviser(L):
    n = len(L)
    milieu = n//2
    return L[:milieu], L[milieu:]
```

on peut par exemple diviser $L = [14,\ 1,\ 4,\ 3]$ avec:

[2]:
```
L = [14, 1, 4, 3]
print(diviser(L))
```

```
([14, 1], [4, 3])
```

Pour fusionner les deux listes déjà triées L_1 et L_2 on parcourt L_1 et L_2 avec deux indices i et j. On compare les éléments L_1[i] et L_2[j] et on rempli la liste à retourner avec le plus petit élément et on incrémente l'indice correspondant.

[3]:
```
def fusion(L_1,L_2) :
    i=0
    j=0
    L = []
    while (i<len(L_1))and(j<len(L_2)):
        if L_1[i]<L_2[j] :
            L.append(L_1[i])
            i += 1
        else :
            L.append(L_2[j])
            j +=1
    if i == len(L_1) :
        L += L_2[j:]
    elif j == len(L_2) :
        L += L_1[i:]
    return L
```

On peut par exemple fusionner L_1=[1, 14] et L_2=[3, 4]

[4]:
```
L_1 = [1, 14]
L_2 = [3, 4]
print(fusion(L_1,L_2))
```

```
[1, 3, 4, 14]
```

La fonction tri fusion devient avec le cas d'arrêt taille de la liste égal à 1

[5]:
```
def tri_fusion(L):
    if len(L) == 1:
        return L
    else:
        L_1,L_2 = diviser(L)
        L_1 = tri_fusion(L_1)
        L_2 = tri_fusion(L_2)
        return fusion(L_1,L_2)
```

par exemple on peut visualiser les différentes étapes du tri de la liste [14 1 4 3] avec les lignes de commandes suivantes

[6]:
```
L = [14, 1, 4, 3]

def tri_fusion_visualisation(L):
    if len(L) == 1:
        return L
    else:
        L_1,L_2 = diviser(L)
        print(L_1,L_2)
        L_1 = tri_fusion_visualisation(L_1)
        L_2 = tri_fusion_visualisation(L_2)
        print(L_1,L_2)
        return fusion(L_1,L_2)

tri_fusion_visualisation(L)
```

```
[14, 1] [4, 3]
[14] [1]
[14] [1]
[4] [3]
[4] [3]
[1, 14] [3, 4]
```

[6]:
```
[1, 3, 4, 14]
```

4 Complexité temporelle

Cette fois-ci la complexité temporelle du tri par fusion ne dépend pas de l'arrangement initial de la liste à trier L. Il n'y a pas de compléxité dans le pire des cas ou dans le meilleur des cas, car l'algorithme effectue toujours le même nombre d'opération pour une taille n de la liste L.

On peut le montrer en calculant la complexité du tri. Soit $C(n)$ la complexité d'un appel de la fonction tri pour une liste de taille n. Lors d'un appel de la fonction tri on appelle : - la fonction diviser de complexité 1, - deux fois la fonction tri pour des listes de taille $n/2$ de complexité $C(n/2)$, - et la fonction fusion qui parcour les deux listes de taille n/2 une fois donc est de complexité $n/2 + n/2 = n$

donc $C(n) = 1 + 2C(n/2) + n$

posons k tel que $n = 2^k$

donc $C(2^k) = 1 + 2C(2^{k-1}) + 2^k$

donc $C(k) = 1 + 2C(k-1) + 2^k = 1 + 2 + 4C(k-2) + 2^k + 2^k = \sum_0^{k-1} 2^i + 2^k C(1) + k2^k = 2^k(1 + C(1) + k)$

or $k = \log_2(n)$ donc $C(n) = n(1 + C(1) + \log_2(n)) = O(n\log(n))$ c'est une complexité quasi-linéaire

5 Exercices

5.1 exercice : absence de meilleur ou pire des cas

- Dérouler chaque étape de l'algorithme de tri par insertion sur la liste $[15, 4, 2, 9, 55, 16, 0, 1]$

[7]:
```
L = [15,4,2,9,55,16,0,1]

def tri_fusion_visualisation(L):
    if len(L) == 1:
        return L
    else:
        L_1,L_2 = diviser(L)
        print(L_1,L_2)
        L_1 = tri_fusion_visualisation(L_1)
        L_2 = tri_fusion_visualisation(L_2)
        print(L_1,L_2)
        return fusion(L_1,L_2)

tri_fusion_visualisation(L)
```

```
[15, 4, 2, 9] [55, 16, 0, 1]
[15, 4] [2, 9]
[15] [4]
[15] [4]
[2] [9]
[2] [9]
[4, 15] [2, 9]
[55, 16] [0, 1]
[55] [16]
[55] [16]
[0] [1]
[0] [1]
[16, 55] [0, 1]
[2, 4, 9, 15] [0, 1, 16, 55]
```

[7]: `[0, 1, 2, 4, 9, 15, 16, 55]`

- Montrer qu'il n'y a pas de meilleur ou pire des cas en déroulant chaque étape du tri sur une liste ordonné ou dans l'ordre inverse.

[8]:
```
L = [0, 1, 2, 4, 9, 15, 16, 55]

def tri_fusion_visualisation(L):
    if len(L) == 1:
        return L
    else:
        L_1,L_2 = diviser(L)
        print(L_1,L_2)
        L_1 = tri_fusion_visualisation(L_1)
        L_2 = tri_fusion_visualisation(L_2)
        print(L_1,L_2)
        return fusion(L_1,L_2)

tri_fusion_visualisation(L)
```

```
[0, 1, 2, 4] [9, 15, 16, 55]
[0, 1] [2, 4]
[0] [1]
[0] [1]
[2] [4]
[2] [4]
[0, 1] [2, 4]
[9, 15] [16, 55]
[9] [15]
[9] [15]
[16] [55]
[16] [55]
[9, 15] [16, 55]
[0, 1, 2, 4] [9, 15, 16, 55]
```

[8]: `[0, 1, 2, 4, 9, 15, 16, 55]`

[9]:
```
L = [55, 16, 15, 9, 4, 2, 1, 0]

def tri_fusion_visualisation(L):
    if len(L) == 1:
        return L
    else:
        L_1,L_2 = diviser(L)
        print(L_1,L_2)
        L_1 = tri_fusion_visualisation(L_1)
        L_2 = tri_fusion_visualisation(L_2)
        print(L_1,L_2)
```

```
        return fusion(L_1,L_2)

tri_fusion_visualisation(L)
```

```
[55, 16, 15, 9] [4, 2, 1, 0]
[55, 16] [15, 9]
[55] [16]
[55] [16]
[15] [9]
[15] [9]
[16, 55] [9, 15]
[4, 2] [1, 0]
[4] [2]
[4] [2]
[1] [0]
[1] [0]
[2, 4] [0, 1]
[9, 15, 16, 55] [0, 1, 2, 4]
```

[9]:
```
[0, 1, 2, 4, 9, 15, 16, 55]
```

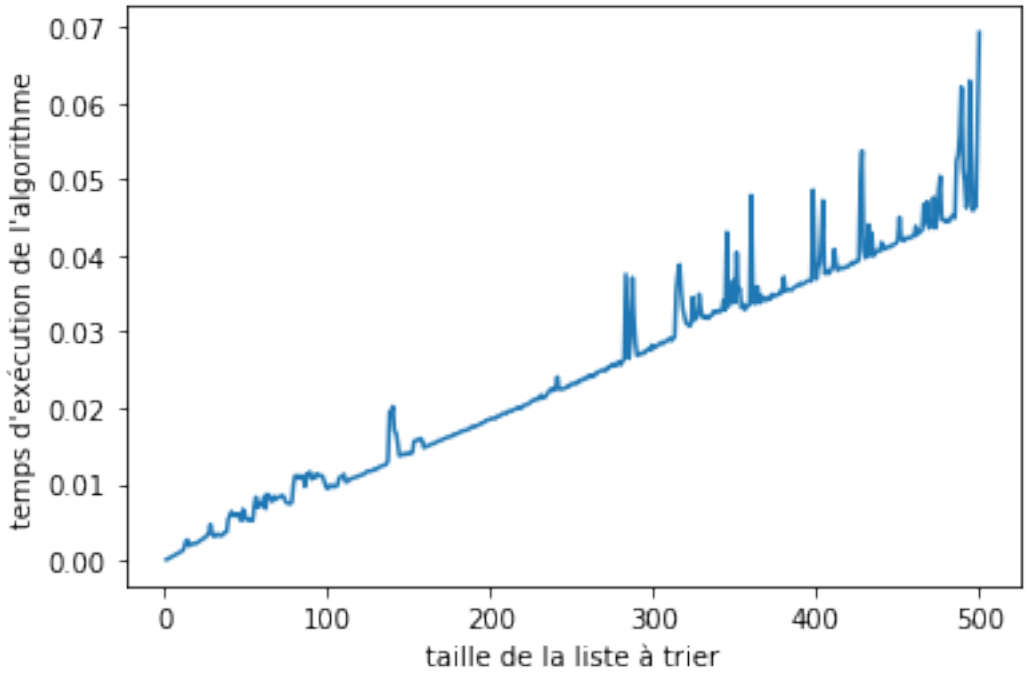
5.2 exercice : calcul numérique de la complexité temporelle

- à l'aide de tirage au sort successif de liste de différentes tailles, mesurer numériquement le temps mis pour l'exécution du tri par fusion sur ces listes et tracer le temps d'éxécution en fonction de la taille des listes à trier.

[23]:
```
import time
import matplotlib.pyplot as plt
import numpy as np
import random as rd

i_max = 500

j_max = 20

duree = []
taille = []
for i in range(i_max):
    taille.append(i+1)
    duree.append(0)
    L=np.arange(i+1)
    for j in range(j_max):
        rd.shuffle(L)
        depart = time.clock()
        tri_fusion(L)
        arrive = time.clock()
```

```
        duree[i]=duree[i]+arrive-depart

plt.plot(taille,duree,'-')
plt.xlabel('taille de la liste à trier')
plt.ylabel("temps d'exécution de l'algorithme")
plt.show()
```

```
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:18: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:20: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
```



- est-ce cohérent avec le calcul de complexité effectué précédemment ?

On observe bien un résultat quasi-linéaire.

5.3 exercice : complexité spatiale

- calculer la complexité spatiale d'un algorithme de tri par fusion

à chaque appel récursif on doit créer les espaces mémoires pour trier L_1 et L_2, puis on créer une liste dans fusion qui fait la taille de L_1+L_2 d'où

$C(n) = C(n/2) + C(n/2) + n$

on a la même relation de récurrence que pour la compléxité temporelle donc $C(n) = O(n \log n)$ c'est une complexité quasi-linéaire.

5.4 exercice : classement

Plusieurs joueurs ont joué au pendu, nous disposons d'une liste de scores contenant les noms et scores (le taux de réussite) de chaque joueur. La liste est de cette forme : [['Marc', 0.87], ['Maryam', 0.99], ['Jean-Loup', 0.91], ['Hubert', 0.84]].

- Écrire une fonction tri_score(L) qui trie une liste de scores par score décroissant, en utilisant le tri fusion.

[11]:
```
def diviser_score(L):
    n = len(L)
    milieu = n//2
    return L[:milieu], L[milieu:]

def fusion_score(L_1,L_2) :
    i=0
    j=0
    L = []
    while (i<len(L_1))and(j<len(L_2)):
        if L_1[i][1]<L_2[j][1] : # ce sont les mêmes programmes sauf pour la␣
 ↪comparaison, on ne compare que le score
            L.append(L_1[i])
            i += 1
        else :
            L.append(L_2[j])
            j +=1
    if i == len(L_1) :
        L += L_2[j:]
    elif j == len(L_2) :
        L += L_1[i:]
    return L

def tri_score(L):
    if len(L) == 1:
        return L
    else:
        L_1,L_2 = diviser_score(L)
        L_1 = tri_score(L_1)
        L_2 = tri_score(L_2)
        return fusion_score(L_1,L_2)
```

```
L = [['Marc', 0.87], ['Maryam', 0.99], ['Jean-Loup', 0.91], ['Hubert', 0.84]]
tri_score(L)
```

[11]:
```
[['Hubert', 0.84], ['Marc', 0.87], ['Jean-Loup', 0.91], ['Maryam', 0.99]]
```

5.5 exercice : optimisation du tri par fusion

Une façon d'optimiser le tri par fusion consiste à éviter la fusion lorsque, à l'issue des deux appels récursifs, les éléments d'une liste se trouvent être tous plus petits que les éléments de l'autre moitié. On le teste facilement en comparant l'élément le plus grand d'une moitié et l'élément le plus petit de l'autre.

- Modifier le tri fusion en suivant cette idée.

[12]:
```
def diviser_optimisation(L):
    n = len(L)
    milieu = n//2
    return L[:milieu], L[milieu:]

def fusion_optimisation(L_1,L_2) :
    i=0
    j=0
    L = []
    while (i<len(L_1))and(j<len(L_2)):
        if L_1[i]<L_2[j] :
            L.append(L_1[i])
            i += 1
        else :
            L.append(L_2[j])
            j +=1
    if i == len(L_1) :
        L += L_2[j:]
    elif j == len(L_2) :
        L += L_1[i:]
    return L

def tri_fusion_optimisation(L):
    if len(L) == 1:
        return L
    else:

        L_1,L_2 = diviser_optimisation(L)

        L_1 = tri_fusion_optimisation(L_1)
        L_2 = tri_fusion_optimisation(L_2)

        if L_1[-1] <= L_2[0] :
            return L_1+L_2
```

```
        elif L_2[-1] <= L_1[0] :
            return L_2+L_1
        else :
            return fusion_optimisation(L_1,L_2)
```

On teste si le nouveau tri fusion fonctionne correctement.

[13]:
```
L = [15,4,2,9,55,16,0,1]
tri_fusion_optimisation(L)
```

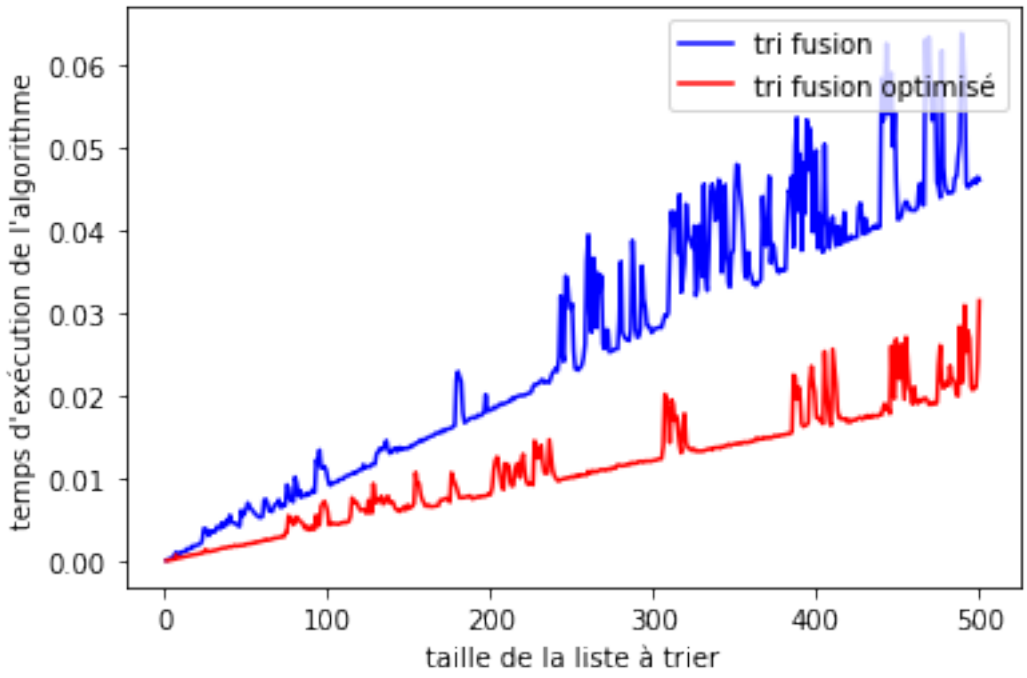
[13]:
```
[0, 1, 2, 4, 9, 15, 16, 55]
```

On compare les temps de calcul du tri fusion initial et optimisé

[14]:
```
import time
import matplotlib.pyplot as plt
import numpy as np
import random as rd

i_max = 500

j_max = 20

duree = []
taille = []
for i in range(i_max):
    taille.append(i+1)
    duree.append(0)
    L=np.arange(i+1)
    for j in range(j_max):
        rd.shuffle(L)
        depart = time.clock()
        tri_fusion(L)
        arrive = time.clock()
        duree[i]=duree[i]+arrive-depart

duree_optimisation = []
taille_optimisation = []
for i in range(i_max):
    taille_optimisation.append(i+1)
    duree_optimisation.append(0)
    L_optimisation=np.arange(i+1)
    for j in range(j_max):
        rd.shuffle(L_optimisation)
        depart = time.clock()
        tri_fusion_optimisation(L_optimisation)
        arrive = time.clock()
        duree_optimisation[i]=duree_optimisation[i]+arrive-depart

plt.plot(taille,duree,'-b')
```

```
plt.plot(taille_optimisation,duree_optimisation,'-r')
plt.xlabel('taille de la liste à trier')
plt.ylabel("temps d'exécution de l'algorithme")
plt.legend(('tri fusion','tri fusion optimisé'), loc='upper right')
plt.show()
```

```
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:18: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:20: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:31: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:33: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
```



5.6 exercice : accélération par hybridation tri par insertion et fusion

Une idée classique pour accélérer un algorithme de tri consiste à effectuer un tri par insertion quand le nombre d'éléments à trier est petit, c'est-à-dire devient inférieur à une constante fixée à l'avance (par exemple 5).

- Modifier le tri par fusion pour prendre en compte cette idée. On pourra reprendre la fonction de tri par insertion.

[15]:
```
def echange(L, i, j):
    L[i], L[j] = L[j], L[i]

def insertion(L, i):
    j = i
    while j > 0 and L[j] < L[j - 1]:
        echange(L, j - 1, j)
        j -= 1

def tri_insertion(L):
    L_prime = []
    L_prime.append(L[0])
    for i in range(1, len(L)):
        L_prime.append(L[i])
        insertion(L_prime, i)
    return L_prime

def diviser_acceleration(L):
    n = len(L)
    milieu = n//2
    return L[:milieu], L[milieu:]

def fusion_acceleration(L_1,L_2):
    i=0
    j=0
    L = []
    while (i<len(L_1))and(j<len(L_2)):
        if L_1[i]<L_2[j] :
            L.append(L_1[i])
            i += 1
        else :
            L.append(L_2[j])
            j +=1
    if i == len(L_1) :
        L += L_2[j:]
    elif j == len(L_2) :
        L += L_1[i:]
    return L
```

```
def tri_fusion_acceleration(L):
    if len(L) <= 5:
        return tri_insertion(L)
    else:
        L_1,L_2 = diviser_acceleration(L)
        L_1 = tri_fusion_acceleration(L_1)
        L_2 = tri_fusion_acceleration(L_2)
        return fusion_acceleration(L_1,L_2)
```

On teste si le nouveau tri fusion fonctionne correctement.

[16]:
```
L = [15,4,2,9,55,16,0,1]
tri_fusion_acceleration(L)
```

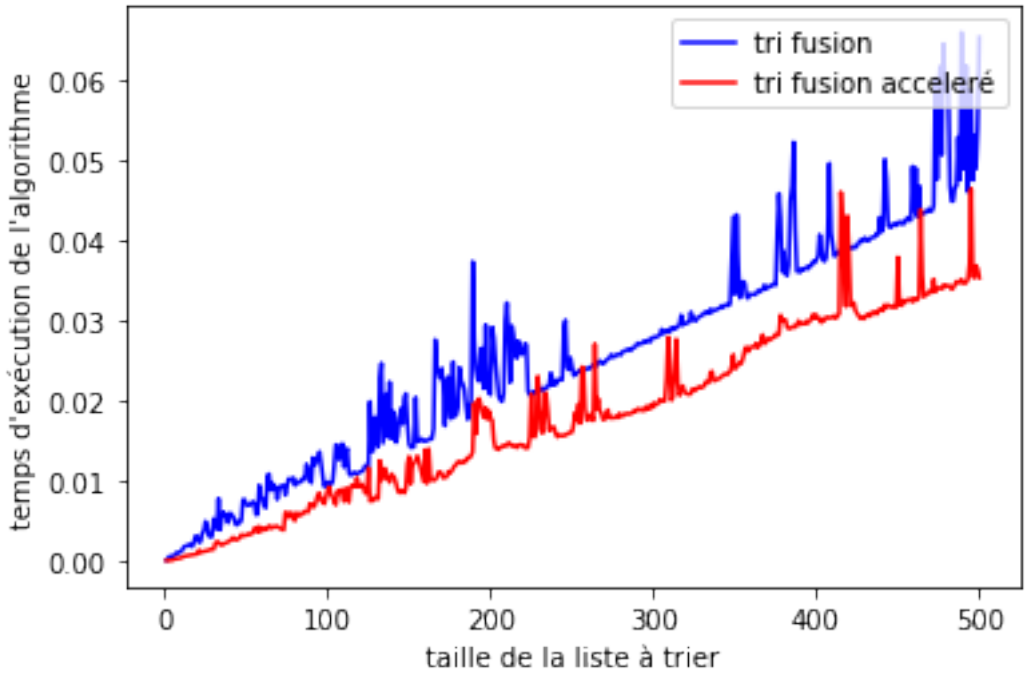
[16]:
```
[0, 1, 2, 4, 9, 15, 16, 55]
```

On compare les temps de calcul du tri fusion initial et acceleré

[17]:
```
import time
import matplotlib.pyplot as plt
import numpy as np
import random as rd

i_max = 500

j_max = 20

duree = []
taille = []
for i in range(i_max):
    taille.append(i+1)
    duree.append(0)
    L=np.arange(i+1)
    for j in range(j_max):
        rd.shuffle(L)
        depart = time.clock()
        tri_fusion(L)
        arrive = time.clock()
        duree[i]=duree[i]+arrive-depart

duree_acceleration = []
taille_acceleration = []
for i in range(i_max):
    taille_acceleration.append(i+1)
    duree_acceleration.append(0)
    L_acceleration=np.arange(i+1)
    for j in range(j_max):
        rd.shuffle(L_acceleration)
        depart = time.clock()
        tri_fusion_acceleration(L_acceleration)
```

```
        arrive = time.clock()
        duree_acceleration[i]=duree_acceleration[i]+arrive-depart

plt.plot(taille,duree,'-b')
plt.plot(taille_acceleration,duree_acceleration,'-r')
plt.xlabel('taille de la liste à trier')
plt.ylabel("temps d'exécution de l'algorithme")
plt.legend(('tri fusion','tri fusion acceleré'), loc='upper right')
plt.show()
```

```
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:18: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:20: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:31: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
C:\Users\remib\AppData\Local\Continuum\anaconda3\lib\site-
packages\ipykernel_launcher.py:33: DeprecationWarning: time.clock has been
deprecated in Python 3.3 and will be removed from Python 3.8: use
time.perf_counter or time.process_time instead
```



5.7 exercice : tri fusion en place

- Modifier l'algorithme de tri par fusion pour obtenir un tri en place, sans créer toutes les listes auxiliaires.

[18]:
```
def diviser_en_place(debut,fin):
    n = fin-debut+1
    milieu = n//2+debut
    return (debut,milieu,milieu,fin)

def fusion_en_place(L,debut_1,fin_1,debut_2,fin_2):
    i=debut_1
    j=debut_2
    while (i<fin_1)and(j<fin_2):
        if L[i]<L[j] :
            i += 1
        else :
            L = L[:i]+[L[j]]+L[i:j]+L[j+1:]
            j +=1
            i +=1
            fin_1 +=1
    return L

def tri_fusion_en_place_rec(L,debut,fin):
    if debut+1 >= fin :
        return L
    else :
        debut_1,fin_1,debut_2,fin_2 = diviser_en_place(debut,fin)
        L = tri_fusion_en_place_rec(L,debut_1,fin_1)
        L = tri_fusion_en_place_rec(L,debut_2,fin_2)
        L = fusion_en_place(L,debut_1,fin_1,debut_2,fin_2)
        return L

def tri_fusion_en_place(L):
    return tri_fusion_en_place_rec(L,0,len(L))
```

[19]:
```
L = [15,4,2,9,55,16,0,1]
tri_fusion_en_place(L)
```

[19]:
```
[0, 1, 2, 4, 9, 15, 16, 55]
```

5.8 exercice : tri fusion intératif

- Modifier l'algorithme de tri par fusion pour obtenir une version itérative du tri par fusion.

[20]:
```
def tri_fusion_iteratif(L):
    n = len(L)
    n_prime = 1
    while n_prime <= n:
        i_max = n//n_prime
        for i in range(0,i_max):
            debut_1 = i*n_prime
            fin_1 = i*n_prime+n_prime//2
            debut_2 = fin_1
            fin_2 = (i+1)*n_prime
            L = fusion_en_place(L,debut_1,fin_1,debut_2,fin_2)
        debut_1 = (i_max-1)*n_prime
        fin_1 = ((i_max-1)*n_prime+n)//2
        debut_2 = fin_1
        fin_2 = n
        L = fusion_en_place(L,debut_1,fin_1,debut_2,fin_2)
        n_prime = n_prime*2
    return L

L = [15,4,2,9,55,16,0,1]
tri_fusion_iteratif(L)
```

[20]:
```
[0, 1, 2, 4, 9, 15, 16, 55]
```