Info 5 : Algorithmes de tri

Contents

Préambule :

[50]:
```
import matplotlib.pyplot as plt
from random import randint
import timeit
import numpy as np
```

1 Un peu de théorie...

On considère une liste de n nombres, on peut la générer aléatoirement grâce à L=[randint(0,k) for i in range(n)] . Cette commande génère une liste de n entiers aléatoires, compris entre 0 et k (inclus).

[51]:
```
def liste_al(k, n):
    return [randint(0,k) for i in range(n)]

L=liste_al(20, 20)
print(L)
L.sort()
print(L)
```

```
[12, 12, 20, 7, 13, 10, 3, 16, 4, 0, 11, 13, 14, 8, 0, 6, 2, 14, 12, 20]
[0, 0, 2, 3, 4, 6, 7, 8, 10, 11, 12, 12, 12, 13, 13, 14, 14, 16, 20, 20]
```

Le travail de cette séance va consister à trier une liste, du nombre le plus petit au plus grand. L'intérêt d'une telle opération est d'obtenir une liste plus facilement manipulable. Par exemple, il

est beaucoup plus facile (et rapide !) de rechercher un élément donné si la liste est déjà triée : on peut, par exemple, faire une recherche par dichotomie (voir séance de révision). C'est le cas d'une bibliothèque : les livres sont (normalement) triés pour permettre de trouver rapidement un livre !

Différents algorithmes de tri seront présentés, on s'interessera plus particulièrement, de façon naturelle, à leur efficacité. Ainsi, on essayera de donner la compléxité de ces algorithme pour trier une liste de N éléments, et on donnera des évaluations numériques du temps nécessaire pour trier une liste.

Dans la vidéo *pyramid* on voit, de façon imagée, le tri d'une telle liste (chaque nombre est représenté par une barre dont la longueur est proportionnelle à ce nombre).

Deux fonctions python proposent déjà nativement de trier une liste : `L.sort()` et `sorted(L)`. La première modifie la liste L, alors que la seconde non (on peut ainsi garder la liste initiale inchangée). Évidemment, notre but ici est coder par nous-même ces fonctions, afin de comprendre leur fonctionnement !

2 Algorithmes de tri "naifs"

Dans le cadre du programme, nous allons étudier plusieurs algorithmes de tri, d'efficacités variées. Le site https://professeurb.github.io/articles/tris/ présente des animations de ces différents algorithmes.

2.1 Un tri spécifique

On considère une liste `L` contenant n nombres entiers, chaque entier étant compris entre 0 et k (entier) inclus. C'est exactement ce type de liste que la fonction `liste_al(k, n)` renvoie. Nous allons écrire une fonction permettant de trier cette liste.

Question 1 Écrire une fonction préliminaire `comptage(L,k)` qui renvoie une liste `Lcompt` de longueur `k+1` telle que `Lcomp[i]` soit égal au nombre d'éléments de `L` égaux à l'entier `i`. Tester cette fonction en utilisant sur une liste générée grâce à la fonction `liste_al(k, n)`.

[4]:
```
L = liste_al(10, 20)
print(L)
```

```
[10, 0, 8, 4, 4, 6, 7, 1, 10, 2, 9, 10, 6, 9, 10, 5, 3, 2, 10, 0]
```

[52]:
```
def comptage(L, k):
    Lcomp=[]

    for i in range(k+1):
        cpt = 0
        for j in range(len(L)):
            if L[j] == i:
                cpt += 1
        Lcomp.append(cpt)
    return Lcomp

# Version bien plus efficace :
```

```python
def comptage(L, k):
    Lcomp=[0]*(k+1)
    for elem in L:
        Lcomp[elem] += 1
    return Lcomp
```

[6]:
```python
comptage(L, 10)
```

[6]:
```
[2, 1, 2, 1, 2, 1, 2, 1, 1, 2, 5]
```

Question 2 Utiliser la fonction précédente pour écrire une fonction `Tri_comptage(L,k)` retournant une nouvelle liste `LT`, liste contenant les mêmes nombres que `L`, mais triés dans l'ordre croissant. Tester cette fonction en utilisant sur une liste générée grâce à la fonction `liste_al(k, n)`.

[53]:
```python
def Tri_comptage(L, k):
    LT = [] # liste triée
    Lcomp = comptage(L, k)
    for i in range(k+1):
        # Nombre de fois qu'apparait l'entier i dans la liste L :
        n = Lcomp[i]
        # On l'ajoute donc n fois dans LT :
        for j in range(n):
            LT.append(i)
    return LT

# Autre version :
def Tri_comptage(L, k):
    LT = [] # liste triée
    Lcomp = comptage(L, k)
    for i in range(k+1):
        # Nombre de fois qu'apparait l'entier i dans la liste L :
        n = Lcomp[i]
        # On l'ajoute donc n fois dans LT :
        LT = LT + [i] * n
    return LT
```

[8]:
```python
print(L)
print(Tri_comptage(L,10))
```

```
[10, 0, 8, 4, 4, 6, 7, 1, 10, 2, 9, 10, 6, 9, 10, 5, 3, 2, 10, 0]
[0, 0, 1, 2, 2, 3, 4, 4, 5, 6, 6, 7, 8, 9, 9, 10, 10, 10, 10, 10]
```

Ce tri par comptage est spécifique car il n'est efficace que si les nombres contenus dans la liste sont redandants : cela revient à dire que dans notre exemple, il est d'autant plus efficace que k est petit comparé à n. Il s'agit aussi d'un tri qui n'est pas *en place*, c'est-à-dire quon ne trie et modifie pas directement la liste `L` (qui reste inchangée), mais on créer une *nouvelle* liste triée.

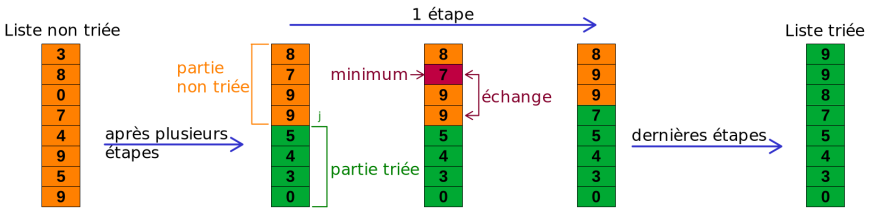
2.2 Un algorithme très naïf : le tri par sélection

Pour trier une liste de nombres plus *générale* que le cas spécifique précédent, la première idée qui peut nous venir à l'esprit est sûrement la suivante :

- on cherche dans la liste le minimum : on le déplace alors en première position : il s'agit de sa position finale, qu'il conservera dans la liste définitivement triée;
- on cherche dans la liste, privée de ce premier élément, le nouveau minimum, qu'on déplace en deuxième position (sa position finale);
- et ainsi de suite...

Pour une liste de taille n , cet algorithme consiste en n étapes. A l'étape j, (j-1) éléments ont déjà étés triés, ils occupent les (j-1) premières positions de la liste. Il faut alors rechercher, dans les (n-j+1) éléments restant le minimum, et le déplacer à la jième position.

Voici une illustration du fonctionnement de cet algorithme de tri, sur un exemple :



Remarque : pour déplacer un élément à la jième position, on peut inverser sa position initiale avec l'élément placé initialement à la jième position.

Pour commencer, on peut déjà, pour des raisons de lisibilité, coder une fonction minimum.

Question 3 : Écrire la fonction `minimum(L,j)` retournant la position du minimum de la liste L, à partir de l'indice j (inclus) :

[54]:
```
def minimum(L,j):
    n = len(L)
    pos = j # on commence en considérant le premier élément de la partie de la␣
 ↪liste restant à trier
    m = L[pos] # m minimum (pour l'instant la valeur du premier élément de la␣
 ↪liste restant à trier)
    for i in range(j+1,n) : # on parcourt le reste de la liste
        if L[i] < m : # si l'élément observé est plus petit que m
            pos = i # on stocke sa position
            m = L[pos] # et il devient le nouveau minimum
    return pos
```

Verifions son fonctionnement :

[10]:
```
L=liste_al(15, 15)
print(L)
print(minimum(L,5))
```

```
[15, 1, 10, 1, 4, 5, 14, 3, 13, 6, 10, 1, 0, 5, 4]
12
```

Question 4 : Compléter alors la fonction `tri_selection(L)` retourner la liste issue de L, mais triée dans l'ordre croissant.

[55]:
```
def tri_selection(L):
    n = len(L)
    # on parcourt la liste (pas la peine d'aller au dernier élément si les␣
 ↪autres sont triés !):
    for j in range(n-1):
            # on cherche la position du minimum de la partie de la liste pas␣
 ↪encore triée :
            pos = minimum(L,j)
            # on inverse ce minimum avec l'élément à la position j :
            L[j], L[pos] = L[pos], L[j]
    return L
```

Verifions son fonctionnement :

[12]:
```
L=liste_al(15, 15)
print(L)
print(tri_selection(L))
```

```
[10, 3, 5, 10, 5, 8, 3, 11, 9, 14, 2, 3, 0, 8, 6]
[0, 2, 3, 3, 3, 5, 5, 6, 8, 8, 9, 10, 10, 11, 14]
```

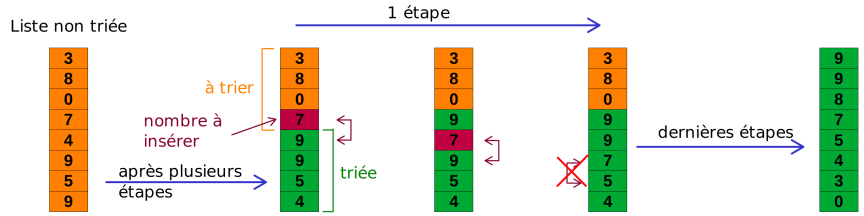
Remarque : la portion de code `return L` *est ici en réalité inutile ici, la liste* `L` *est modifiée chaque fois qu'on agit dessus. Il s'agit donc d'un tri en place*

Analyse de la compléxité : Cette algorithme est de compléxité en $\mathcal{O}(n^2)$. En effet :

- on parcourt une fois la liste -> n "grosses" opérations
- pour chacune de ces grosses opérations, il faut chercher le maximum d'une liste de $n-j$ éléments -> n opérations au maximum, car $n-j$ est dominé par n
- on a donc n opérations dans chacune des n "grosses" opérations, ce qui nous fait n^2 opérations

2.3 Un algorithme un peu moins naïf: le tri par insertion

Le tri par insertion est sans doute le tri qu'on effectue naturellement pour trier un jeu de cartes en même temps qu'on nous le distribue : on sépare la liste en deux parties, la partie déjà triée et l'autre. On considère alors le premier élément de la partie non triée : on recherche où le placer la partie déjà triée.



Attention : la difficulté de cet algorithme est justement d'*insérer* l'élément, il est alors conseillé de décaler tous les autres éléments supérieurs de la partie triée d'une place vers la droite (ou vers le haut sur l'exemple ci-dessus), jusqu'à atteindre la "bonne" place. C'est encore une fois un tri *en place*.

C'est pourquoi nous allons commencer par une fonction d'insertion.

Question 5 : Écrire la fonction `insertion(L,j)`, qui, pour une liste L dont la partie située strictement avant l'indice j est supposée déjà triée, insère l'élément initialement au jième emplacement au "bon" emplacement :

[57]:
```
def insertion(L,j):
    while L[j-1]>L[j] and j>0 :
        # On déplace l'élément initialement à l'emplacement j vers la gauche en␣
 ↪échangeant sa place
        L[j-1],L[j]=L[j],L[j-1]
        # On se déplace vers la gauche dans la liste, en partant de l'indice j
        j-=1
    return L
```

Vérifions son bon fonctionnement :

[16]:
```
L=tri_selection(liste_al(10,10))+liste_al(10,10)
print(L)
print(insertion(L,10))
```

```
[0, 0, 0, 2, 3, 3, 4, 8, 9, 10, 2, 7, 3, 3, 9, 0, 4, 3, 9, 2]
[0, 0, 0, 2, 2, 3, 3, 4, 8, 9, 10, 7, 3, 3, 9, 0, 4, 3, 9, 2]
```

Écrivons alors la fonction de tri par insertion :

Question 6 : Compléter la fonction `tri_insertion(L)` retourner la liste issue de L, mais triée dans l'ordre croissant.

[58]:
```
def tri_insertion(L):
    # On parcourt la liste (pas la peine de commencer au premier élément)
    for i in range(1,len(L)):
        # on déplace l'élément à l'emplacement i vers sa place dans la partie␣
 ↪triée de la liste
        insertion(L,i)
    return L
```

Vérifions son bon fonctionnement :

[18]:
```
L=liste_al(15, 15)
print(L)
print(tri_insertion(L))
```

```
[13, 6, 10, 7, 4, 8, 9, 14, 4, 4, 7, 1, 8, 14, 12]
[1, 4, 4, 4, 6, 7, 7, 8, 8, 9, 10, 12, 13, 14, 14]
```

Analyse de la complexité : Cette algorithme est toujours de complexité en $\mathcal{O}(n^2)$. En effet :

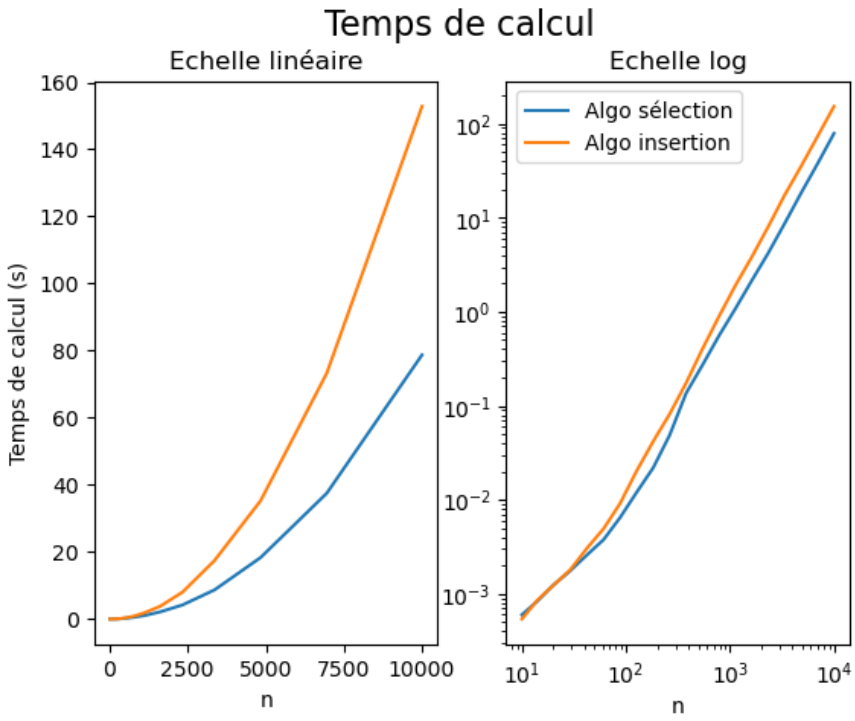
- on parcourt une fois la liste -> n “grosses” opérations
- pour chacune de ces “grosses” opérations -> il faut chercher le bon emplacement dans j emplacements -> n opérations au maximum
- on a donc n opérations dans chacune des n “grosses” opérations, ce qui nous fait n^2 opérations

La complexité est la même que pour le premier algorithme : en effet, les étapes sont en réalités très similaires (on parcourt la liste -> n opérations, et à chaque fois on effectue la comparaion d'un élément avec le n autres -> $n \times n$ opérations). Cependant, cet algorithme est quand même un peu plus rapide :

[59]:
```
Tselection,Tinsertion,N=[],[],[]
for n in np.logspace(1,4,20):
    n = int(n)
    N.append(n)
    Tselection.append(timeit.timeit('tri_selection(liste_al(n,n))',␣
 ↪globals=globals(), number=50))
    Tinsertion.append(timeit.timeit('tri_insertion(liste_al(n,n))',␣
 ↪globals=globals(), number=50))

fig, axs = plt.subplots(1, 2)
axs[0].plot(N,Tselection,label='Algo sélection')
axs[0].plot(N,Tinsertion,label='Algo insertion')
axs[0].set_title('Echelle linéaire')
axs[0].set_xlabel('n')
axs[0].set_ylabel('Temps de calcul (s)')
fig.suptitle("Temps de calcul", fontsize=16)

axs[1].loglog(N,Tselection,label='Algo sélection')
axs[1].loglog(N,Tinsertion,label='Algo insertion')
axs[1].set_title('Echelle log')
axs[1].set_xlabel('n')
plt.legend()
plt.show()
```



La pente en échelle log/log semble la même, ce qui est la signature de la même complexité. On peut même devisner une petnte de 2 en échelle log-log, signature d'une complexité en n^2. L'aglorithme par insertion est cependant, de façon surprenante, un peu plus lent ici… Je pense que cette lenteur est imputable au fait que dans l'écriture de cet algorithme par insertion, on modifie en permanence la liste, opération coûteuse en temps "machine" (mais ce n'est pas un temps de "calculs").

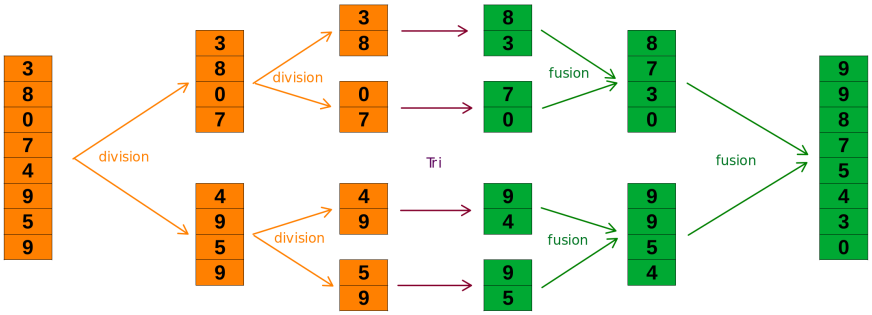
En terme de nombre "d'opérations", les deux algorithme ont la même complexité "moyenne" (en n^2), mais le nombre "exact" d'opérations est généralement plus faible avec l'algorithme par insertion. Cela peut se comprendre avec le cas extrême d'un liste à tirer qui est déjà triée ! Pour le tri par insertion, la complexité sera dans ce cas précis en n, alors que celui par sélection restera en n^2.

3 Un algorithme plus efficace : l'algorithme de tri fusion

La stratégie est basé sur la stratégie "Diviser pour régner". Pour cela, on va:

- diviser la liste initiale en deux listes de même taille (à 1 près, dans le cas d'une liste de taille impaire);
- trier ces deux sous-listes récursivement;
- fusionner les deux listes en une nouvelle liste, triée.

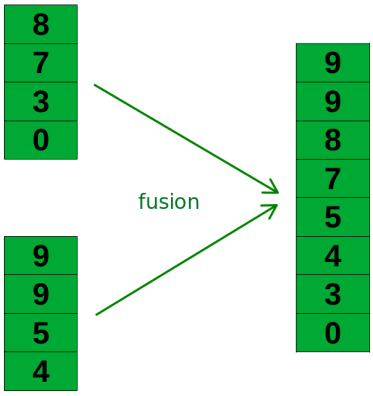
Voici un exemple d'un tel tri :



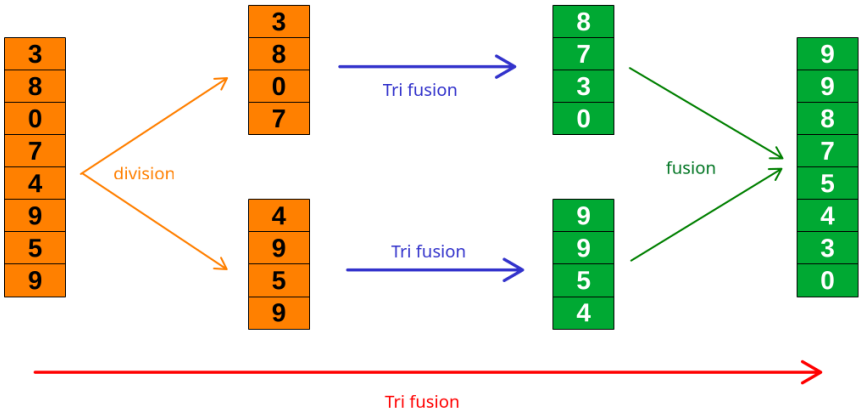
L'étape "difficile" de cet algorithme est naturellement la fusion : il s'agit, à partir de deux listes déjà triées, de réaliser la liste complète triée réunissant ces deux listes.

Pour réaliser ce tri fusion, on écrit deux fonctions :

- La fonction `fusion(L1,L2)` qui prend en argument deux listes triées et renvoie la liste triée obtenue en les réunissant.



- La fonction `tri_fusion(L)`, récursive, qui retourne la liste L triée.



Question 7 : Écrire les codes de ces deux fonctions, puis tester le tri d'une liste de nombres.

[60]:
```
def fusion(L1,L2):
    n1,n2=len(L1),len(L2)
```

```
    i1,i2=0,0
    L=[]
    # Parcours "simultané" de L1 et L2
    while i1<n1 and i2<n2:
        e1,e2=L1[i1],L2[i2]
        if e1<=e2:
            L.append(e1)
            i1=i1+1
        else:
            L.append(e2)
            i2=i2+1
    # On est arrivé au bout de L2 : on rajoute le reste de L1
    if i2==n2:
        L=L+L1[i1:]
    # ou L.extend(L1[i1:])
    # Sinon on rajoute le reste de L2
    else:
        L=L+L2[i2:]
    # ou L.extend(L2[i2:])
    return L
```

Testons :

[25]:
```
L1, L2 = liste_al(20, 10), liste_al(20, 10)

print(L1)
print(tri_selection(L1))
print(L2)
print(tri_selection(L2))
print(fusion(L1, L2))
```

```
[6, 4, 12, 15, 9, 3, 9, 8, 9, 18]
[3, 4, 6, 8, 9, 9, 9, 12, 15, 18]
[10, 16, 11, 11, 12, 11, 15, 10, 11, 17]
[10, 10, 11, 11, 11, 11, 12, 15, 16, 17]
[3, 4, 6, 8, 9, 9, 9, 10, 10, 11, 11, 11, 11, 12, 12, 15, 15, 16, 17, 18]
```

[61]:
```
def tri_fusion(L):
    n=len(L)
    if len(L)<2:
        return L
    else:
        m=n//2
        L1 = L[:m]
        L2 = L[m:]
        return fusion(tri_fusion(L1),tri_fusion(L2))
```

[27]:
```
L = liste_al(15, 15)

print(L,tri_fusion(L))
```

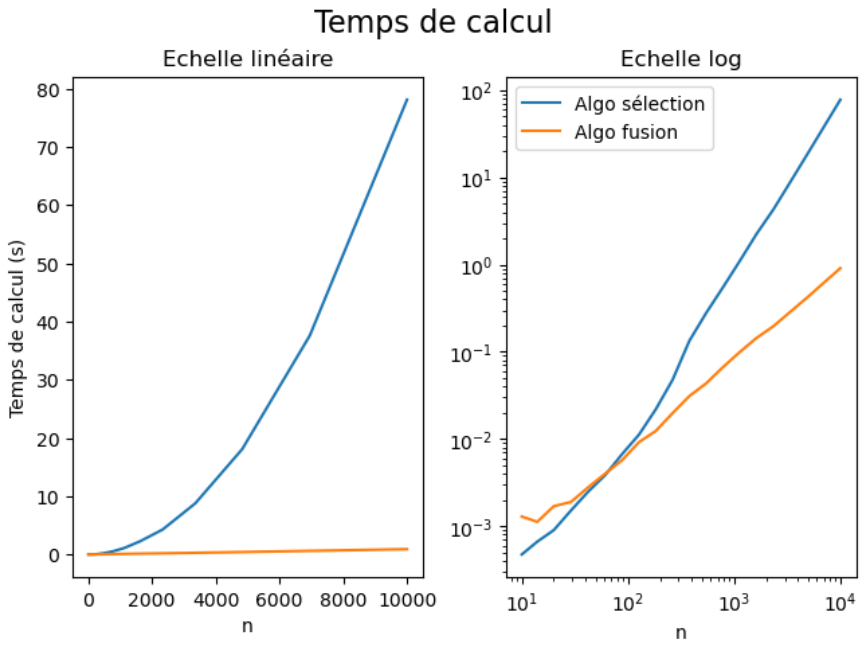
```
[3, 15, 13, 1, 4, 12, 5, 11, 11, 15, 9, 14, 10, 10, 13] [1, 3, 4, 5, 9, 10, 10,
11, 11, 12, 13, 13, 14, 15, 15]
```

La complexité est en $\mathcal{O}(n\ log\ n)$: ce code est bien plus efficace ques les précédent. Vérifions-le, avec des temps de calculs :

[68]:
```
Tselection,Tfusion,N=[],[],[]
for n in np.logspace(1,4,20):
    n = int(n)
    N.append(n)
    Tselection.append(timeit.timeit('tri_selection(liste_al(n,n))',␣
 ↪globals=globals(), number=50))
    Tfusion.append(timeit.timeit('tri_fusion(liste_al(n,n))',␣
 ↪globals=globals(), number=50))

fig, axs = plt.subplots(1, 2, constrained_layout=True)
axs[0].plot(N,Tselection,label='Algo sélection')
axs[0].plot(N,Tfusion,label='Algo fusion')
axs[0].set_title('Echelle linéaire')
axs[0].set_xlabel('n')
axs[0].set_ylabel('Temps de calcul (s)')
fig.suptitle("Temps de calcul", fontsize=16)

axs[1].loglog(N,Tselection,label='Algo sélection')
axs[1].loglog(N,Tfusion,label='Algo fusion')
axs[1].set_title('Echelle log')
axs[1].set_xlabel('n')
plt.legend()
plt.show()
```

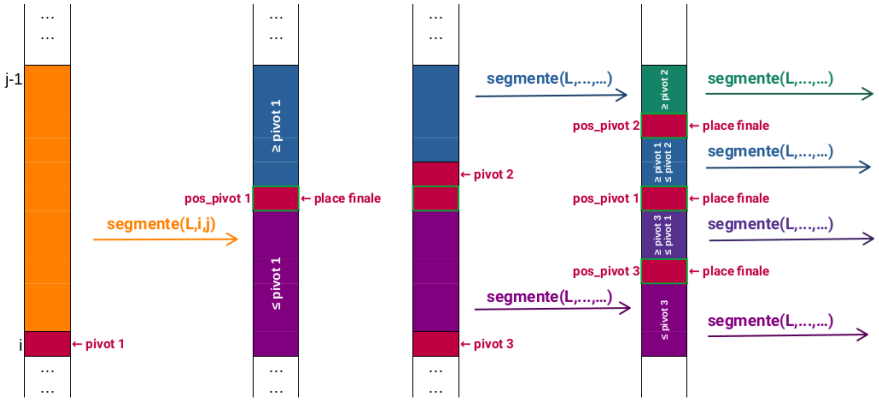


Effectivement, le tri par fusion est bien plus rapide, et plus important, la pente du graphe en échelle log plus faible (peut-être même un comportement non linéaire) : c'est la signature d'une complexité moindre.

4 Un algorithme rapide: le tri "rapide" (quicksort) - Facultatif

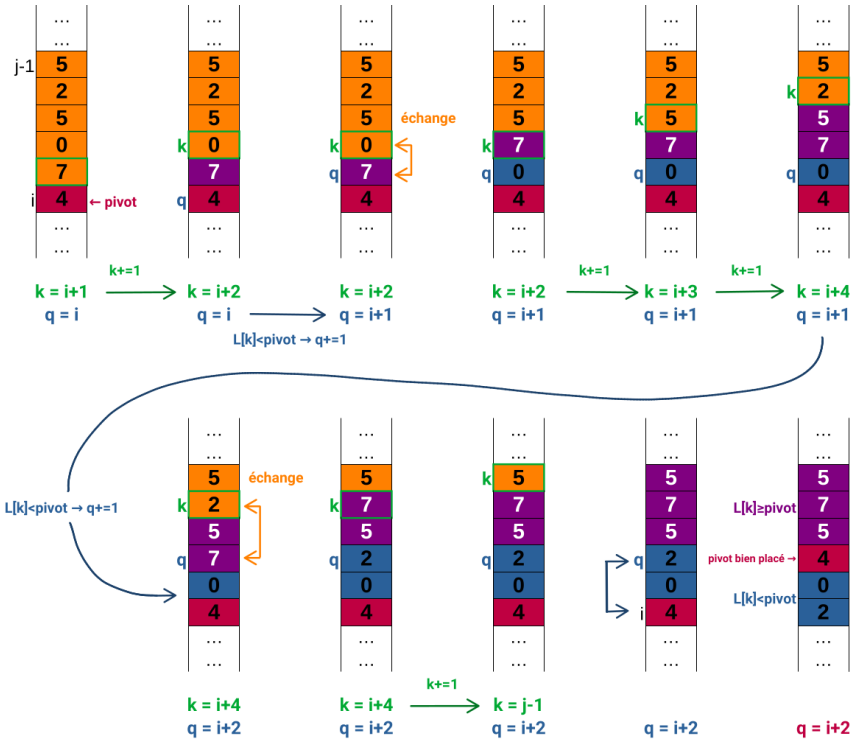
Le tri "rapide" repose sur la stratégie dite de « Diviser pour régner ». Le principe est de partager une liste en 2 sous-listes telles que dans la première, toutes les valeurs prises soient inférieures à celles de la seconde. On a alors décomposé un problème en deux sous-problèmes plus simples. A la fin, on regroupe les résultats de chaque sous-problème pour arriver au résultat. Comme ici le sous-problème est similaire au problème, il est alors possible d'utiliser une fonction récursive pour obtenir, à la fin, une liste triée.

Voici ci-dessous le fonctionnement schématique de l'algorithme. Une partie de la liste, entre l'indice i (inclu) et l'indice j (exclu), est divisée (ou segmentée) en 3 parties, par le fonction `segmente` : cette fonction déplace un élément, nommé pivot, initialement situé en i, de sorte que les éléments situés avant (ou en dessous) soient plus petits que lui, et que les éléments après (ou au-dessus) soient plus grands. On procède ensuite récursivement sur chacune de ces deux parties.



Plus précisement, voici comment procède la fonction `segmente` :

- On choisit dans la partie encore non triée de la liste, comprise entre i et j, un élément particulier, noté *pivot*. Pour simplifier, on prendra toujours le premier élément de cette partie de la liste (4 dans l'exemple ci-dessous, de couleur magenta).
- On déplace les autres éléments de la liste pour faire en sorte que tous les éléments plus petits que le *pivot* se retrouvent *avant* le *pivot*, selon la démarche suivante :



- Il apparait alors deux sous-listes (en bleu et en violet sur l'exemple ci-dessus), de part et d'autre du *pivot*, qu'il faudra par la suite trier selon la même méthode (via une fonction récursive). Le pivot occupe alors sa position définitive.

Sur l'exemple ci-dessus, la variable k représente la position observée dans la liste à triée, et la lettre q la position du dernier élément plus petit que le pivot déjà rencontré ($q \leq k$).

Question 8 : Compléter le code de la fonction `segmente(L,i,j)` qui s'occupe de la partie de la liste L comprise entre i et j-1 (compris), c'est-à-dire qu'elle doit choisir l'élément en i comme pivot, et modifier la liste L en plaçant les éléments compris entre i et j qui sont plus petits que le pivot entre i et l'indice de ce pivot. De même, les éléments compris entre i et j qui sont plus grands que le pivot doivent être placés entre l'indice du pivot et j. Cette fonction retournera la position du pivot.

Pour se simplifier la tâche, comme sur l'illustration précédente, on choisira la variable k pour parcourir la partie de la liste à modifier (entre i+1 et j-1), et q stockera l'indice du dernier élément plus petit que pivot déjà rencontré (qui est aussi la position finale que devra prendre le pivot, celle de l'élément le plus haut en bleu sur le schéma ci-dessus).

[ ]:
```
def segmente(L,i,j):
    # On choisit le pivot
    pivot=L[i]
    # q stocke la position du dernier élément plus petit que le pivot
    q=i
    # On parcourt la sous-liste
    for k in range(i+1,j):
        if L[k]<pivot:
            # Dans le cas ou l'élément est plus petit que le pivot : q augmente
 ↪de 1
            q+=1
            # On échange les éléments
            L[k],L[q]=L[q],L[k]
    # On s'occupe finalement du pivot : on l'échange de place
    L[q],L[i]=L[i],L[q]
    return q
```

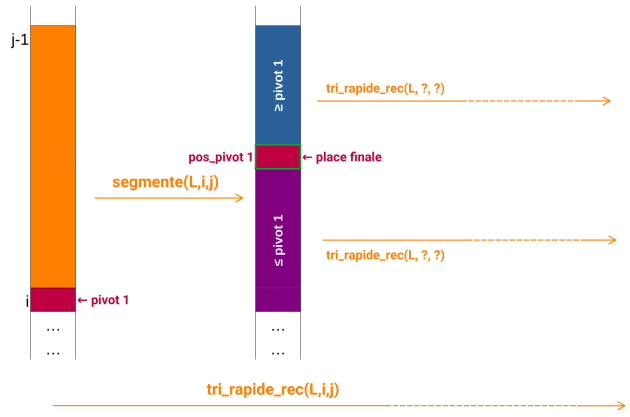
Vérifions :

[30]:
```
L=liste_al(15,15)
print(L)
ppivot=segmente(L,4,15)
print("Position du pivot : ",ppivot," Valeur du pivot : ",L[ppivot])
print(L)
```

```
[6, 6, 0, 13, 9, 14, 6, 2, 7, 12, 11, 6, 8, 15, 0]
Position du pivot :  10  Valeur du pivot :  9
[6, 6, 0, 13, 0, 6, 2, 7, 6, 8, 9, 14, 12, 15, 11]
```

Poursuivons : il faut désormais écrire la fonction qui trie complètement la liste.

Question 9 : Écrire la fonction `tri_rapide_rec(L,i,j)` qui tri la portion située entre i (compris) et j (non compris) de la liste L via la méthode de tri rapide. Pour cela, elle doit être récursive, comme le montre l'illustration ci-dessous :



Il ne faut bien sur pas oublier les “conditions de terminaison”.

[ ]:
```
def tri_rapide_rec(L,i,j): # Tri entre i et j-1 inclus
     if i < j-1: # S'il reste  plus (strictement) que
                 # 1 élément dans la portion à trier i -> j-1
            pos_pivot=segmente(L,i,j) # On segmente, et on
                                      # récupère la position du pivot
            tri_rapide_rec(L,pos_pivot+1,j) # On trie la partie sup
            tri_rapide_rec(L,i,pos_pivot) # On trie la partie inf
```

Et enfin la fonction `tri_rapide(L)` :

[ ]:
```
def tri_rapide(L):
    tri_rapide_rec(L,0,len(L))
    return L
```

Vérifions :

[26]:
```
L=liste_al(15,15)
print(L)
print(tri_rapide(L))
```

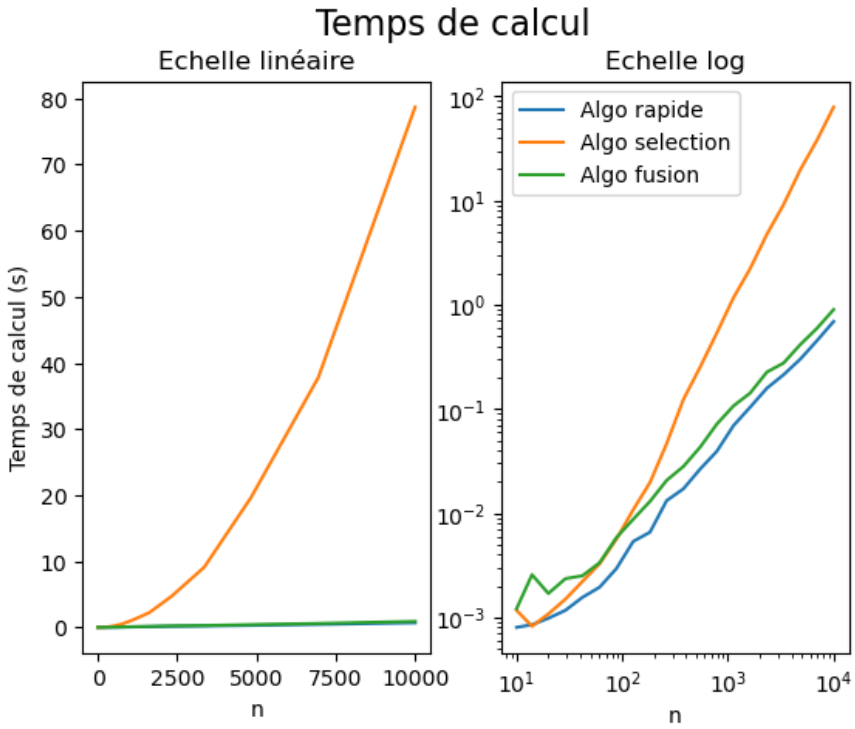
```
[4, 6, 10, 6, 7, 6, 15, 9, 10, 1, 4, 14, 3, 8, 15]
[1, 3, 4, 4, 6, 6, 6, 7, 8, 9, 10, 10, 14, 15, 15]
```

Petit test de rapidité :

[69]:
```
Tfusion, Trapide, Tselection,N=[],[],[],[]
for n in np.logspace(1,4,20):
    n = int(n)
    N.append(n)
    Trapide.append(timeit.timeit('tri_rapide(liste_al(n,n))',␣
 ↪globals=globals(), number=50))
    Tselection.append(timeit.timeit('tri_selection(liste_al(n,n))',␣
 ↪globals=globals(), number=50))
    Tfusion.append(timeit.timeit('tri_fusion(liste_al(n,n))',␣
 ↪globals=globals(), number=50))
```

```
fig, axs = plt.subplots(1, 2)
axs[0].plot(N,Trapide,label='Algo rapide')
axs[0].plot(N,Tselection,label='Algo selection')
axs[0].plot(N,Tfusion,label='Algo fusion')
axs[0].set_title('Echelle linéaire')
axs[0].set_xlabel('n')
axs[0].set_ylabel('Temps de calcul (s)')
fig.suptitle("Temps de calcul", fontsize=16)

axs[1].loglog(N,Trapide,label='Algo rapide')
axs[1].loglog(N,Tselection,label='Algo selection')
axs[1].plot(N,Tfusion,label='Algo fusion')
axs[1].set_title('Echelle log')
axs[1].set_xlabel('n')
plt.legend()
plt.show()
```



Le tri rapide porte bien son nom, étant plus rapide que celui par fusion. Cependant, ils semblent posséder la même compléxité !

Pour aller plus loin :

- On peut écrire, de façon plus plus élégante :

[ ]:
```
def tri_rapide2(L, *args):
    if len(args) == 0:
```

```
        i, j = 0, len(L)
    else:
        i, j = args
    if i + 1 < j:
        pos_pivot = segmente(L, i, j)
        tri_rapide2(L, pos_pivot + 1, j)
        tri_rapide2(L, i, pos_pivot)
    return L
```

- Complexité en $\mathcal{O}(nlog(n))$ au mieux, en $\mathcal{O}(n^2)$ au pire
- Problème de pile qui déborde (*stack overflow*), comme toute fonction qui utilise la récursivité. Pour régler ce problème, et aussi être plus efficace, on peut utiliser une autre fonction de tri pour de "petites" sous-listes. On en profite pour apporter un autre raffinement : on choisit le pivot aléatoirement.

[ ]:
```
def segmente2(L,i,j):
    # On choisit le pivot
    pos_pivot=randint(i,j-1)
    L[i],L[pos_pivot]=L[pos_pivot],L[i]
    pivot=L[i]
    # q stocke la position du dernier élément plus petit que le pivot
    q=i
    # On parcourt la sous-liste
    for k in range(i+1,j):
        if L[k]<pivot:
            # Dans le cas ou l'élément est plus petit que le pivot : q augmente␣
 ↪de 1
            q+=1
            # On échange les éléments
            L[k],L[q]=L[q],L[k]
    # On s'occupe du pivot : on l'échange de place
    L[q],L[i]=L[i],L[q]
    return q

def tri_rapide3(L, *args):
    if len(args) == 0:
        i, j = 0, len(L)
    else:
        i, j = args
    if i + 10 < j:
        pos_pivot = segmente2(L, i, j)
        tri_rapide3(L, pos_pivot + 1, j)
        tri_rapide3(L, i, pos_pivot)
    else:
        L[i:j]=tri_selection(L[i:j])
    return L
```

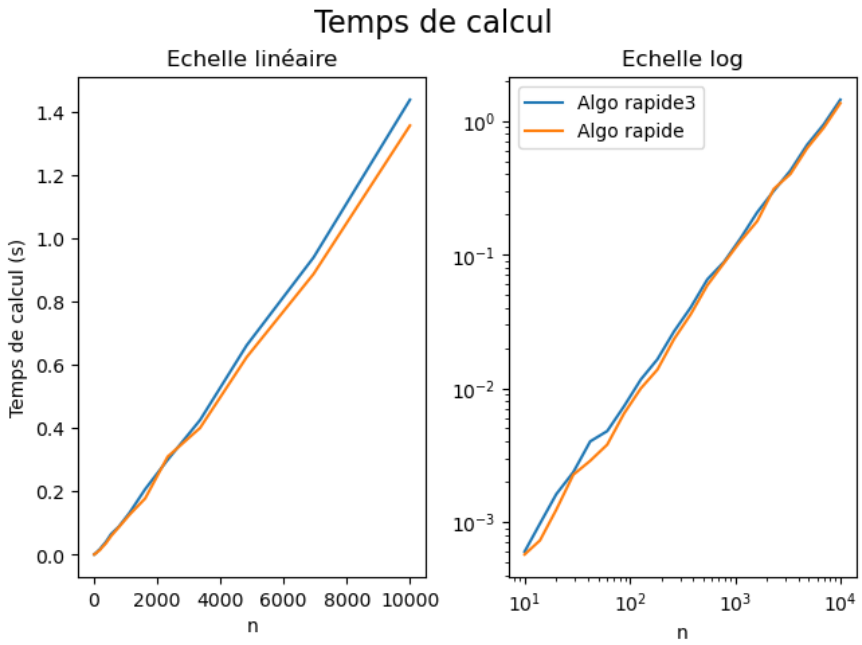
[40]:
```
L=liste_al(30, 30)
print(L)
print(tri_rapide3(L))
```

```
[21, 2, 12, 13, 12, 19, 14, 4, 30, 5, 30, 28, 5, 29, 23, 7, 2, 7, 28, 19, 11,
10, 11, 0, 16, 30, 10, 7, 28, 16]
[0, 2, 2, 4, 5, 5, 7, 7, 7, 10, 10, 11, 11, 12, 12, 13, 14, 16, 16, 19, 19, 21,
23, 28, 28, 28, 29, 30, 30, 30]
```

[70]:
```
Trapide3,Trapide,N=[],[],[]
for n in np.logspace(1,4,20):
    n = int(n)
    N.append(n)
    Trapide.append(timeit.timeit('tri_rapide(liste_al(n,n))',␣
 ↪globals=globals(), number=100))
    Trapide3.append(timeit.timeit('tri_rapide3(liste_al(n,n))',␣
 ↪globals=globals(), number=100))

fig, axs = plt.subplots(1, 2, constrained_layout=True)
axs[0].plot(N,Trapide3,label='Algo rapide3')
axs[0].plot(N,Trapide,label='Algo rapide')
axs[0].set_title('Echelle linéaire')
axs[0].set_xlabel('n')
axs[0].set_ylabel('Temps de calcul (s)')
fig.suptitle("Temps de calcul", fontsize=16)

axs[1].loglog(N,Trapide3,label='Algo rapide3')
axs[1].loglog(N,Trapide,label='Algo rapide')
axs[1].set_title('Echelle log')
axs[1].set_xlabel('n')
plt.legend()
plt.show()
```

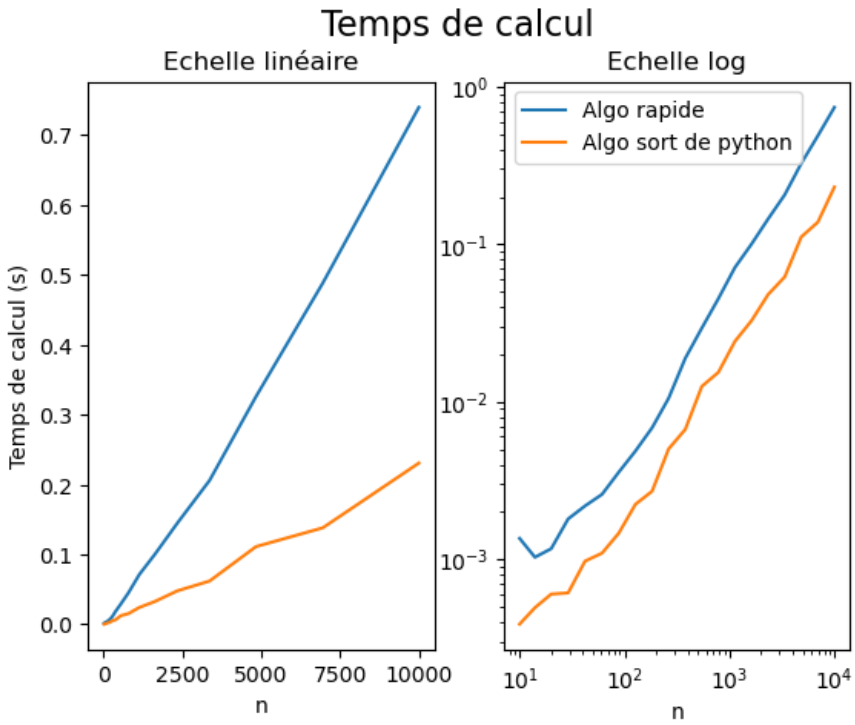


Bon, c'est pas vraiment probant… Et comparons avec l'algo de python :

[73]:
```
Trapide,Tsort,N=[],[],[]
for n in np.logspace(1,4,20):
    n = int(n)
    N.append(n)
    Trapide.append(timeit.timeit('tri_rapide(liste_al(n,n))',␣
 ↪globals=globals(), number=50))
    Tsort.append(timeit.timeit('liste_al(n,n).sort()', globals=globals(),␣
 ↪number=50))

fig, axs = plt.subplots(1, 2)
axs[0].plot(N,Trapide,label='Algo rapide')
axs[0].plot(N,Tsort,label='Algo sort de python')
axs[0].set_title('Echelle linéaire')
axs[0].set_xlabel('n')
axs[0].set_ylabel('Temps de calcul (s)')
fig.suptitle("Temps de calcul", fontsize=16)

axs[1].loglog(N,Trapide,label='Algo rapide')
axs[1].loglog(N,Tsort,label='Algo sort de python')
axs[1].set_title('Echelle log')
axs[1].set_xlabel('n')
plt.legend()
plt.show()
```



Évidemment, l'algorithme proposé par python est ultra-optimisé… Mais la pente en échelle log semble la même : il s'agit sûrement d'une version optimisée du *quicksort*.

5 Résumé en vidéos et un dernier petit exercice…

Voici une vidéo expliquant et comparant les méthodes de tri vues jusqu'à présent :

Et, seulement en dernier recours, pour ceux qu ont encore un peu de mal avec le tri rapide, une vidéo plus folklorique :

Question 10 La médiane d'une liste est la valeur telle qu'il y a autant d'éléments supérieurs et inférieurs à elle. On accepte un "battement" d'une position lorsque la liste contient un nombre impair de termes. En utilisant une des fonctions de tri précédentes, écrire une fonction `f_mediane_normale(L)` qui détermine la médiane d'une liste `L`.

[66]:
```
def f_mediane_normale(L):
    n=len(L)
    tri_rapide(L)
    return L[int(n/2)]
```

[32]:
```
L=liste_al(20,20)
print(L)
print("Médiane : ",f_mediane_normale(L))
print("Liste triée : ",L)
```

```
[13, 2, 9, 19, 15, 8, 3, 8, 4, 11, 15, 4, 5, 12, 3, 7, 6, 9, 20, 6]
Médiane :  8
```

```
Liste triée :  [2, 3, 3, 4, 4, 5, 6, 6, 7, 8, 8, 9, 9, 11, 12, 13, 15, 15, 19,
20]
```

Il existe une méthode moins couteuse qui permet de ne trier que partiellement L et d'obtenir la médiane. Elle consiste à n'utiliser l'algorithme de tri rapide que partiellement. En effet, on a pas besoin de trier toute la liste ! En effet, on recherche le pivot, et on le place à sa place définitive : si ce pivot correspond à la médiane, c'est gagné ! Sinon, s'il se situe au-delà du milieu de la liste, inutile de trier la partie ultérieure de la liste, et inversement. On définit alors la fonction `demi_tri_rapide(L)` permettant de faire cela.

[64]:
```
def demi_tri_rapide_rec(L,i,j):
    n=len(L)
    if i < j-1:
        pos_pivot=segmente(L,i,j)
        if pos_pivot == int(n/2):
            pass
        elif pos_pivot > n/2: # on ne triera alors que la première partie de la␣
 ↪liste
            tri_rapide_rec(L,i,pos_pivot)
        else : # sinon on ne triera alors que la deuxième partie de la liste
            tri_rapide_rec(L,pos_pivot+1,j)

def demi_tri_rapide(L):
    demi_tri_rapide_rec(L,0,len(L))
    return L

def f_mediane_rapide(L):
    n=len(L)
    demi_tri_rapide(L)
    return L[int(n/2)]
```

[34]:
```
L=liste_al(20,20)
print(L)
# Pour vérification du bon fonctionnement :
L2=L.copy()
print("Médiane par méthode rapide: ",f_mediane_rapide(L))
print("Médiane par méthode normale: ",f_mediane_normale(L2))
print("Liste triée : ",L)
```

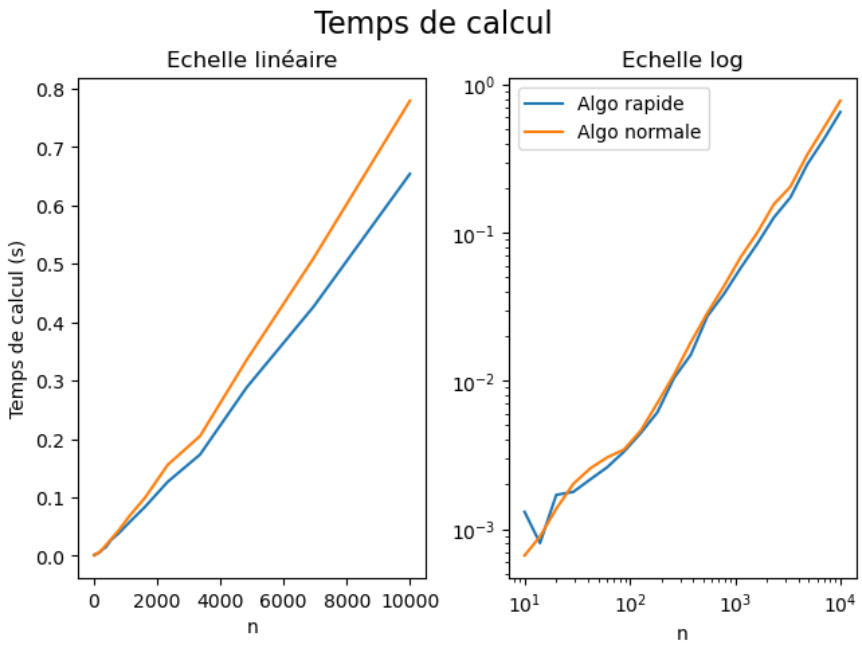
```
[9, 12, 14, 8, 16, 7, 6, 16, 8, 6, 4, 17, 5, 11, 14, 14, 19, 12, 14, 20]
Médiane par méthode rapide:  12
Médiane par méthode normale:  12
Liste triée :  [5, 8, 7, 6, 8, 6, 4, 9, 11, 12, 12, 14, 14, 14, 14, 16, 16, 17,
19, 20]
```

[67]:
```
Trapide,Tnormale,N=[],[],[]
for n in np.logspace(1,4,20):
```

```
    n = int(n)
    N.append(n)
    Trapide.append(timeit.timeit('f_mediane_rapide(liste_al(n,n))',␣
 ↪globals=globals(), number=50))
    Tnormale.append(timeit.timeit('f_mediane_normale(liste_al(n,n))',␣
 ↪globals=globals(), number=50))

fig, axs = plt.subplots(1, 2, constrained_layout=True)
axs[0].plot(N,Trapide,label='Algo rapide')
axs[0].plot(N,Tnormale,label='Algo normale')
axs[0].set_title('Echelle linéaire')
axs[0].set_xlabel('n')
axs[0].set_ylabel('Temps de calcul (s)')
fig.suptitle("Temps de calcul", fontsize=16)

axs[1].loglog(N,Trapide,label='Algo rapide')
axs[1].loglog(N,Tnormale,label='Algo normale')
axs[1].set_title('Echelle log')
axs[1].set_xlabel('n')
plt.legend()
plt.show()
```



On gagne (très peu, mais on gagne !)