

# Info 5 : Tri

## Proposition de correction

### Contents

<b>1 Un peu de théorie...</b>	<b>1</b>
<b>2 Trois algorithmes de tri</b>	<b>2</b>
2.1 Un algorithme très naïf : le tri par sélection . . . . .	2
2.2 Un algorithme un peu moins naïf: le tri par insertion . . . . .	3
2.3 Un algorithme rapide: le tri “rapide” (quicksort) . . . . .	6
2.4 Résumé . . . . .	15
<b>3 Pour aller plus loin</b>	<b>15</b>
3.1 Un dernier algorithme de tri : l’algorithme de tri fusion . . . . .	15
3.2 Utilisation d’un algorithme de tri : détermination de la médiane . . . . .	17

### 1 Un peu de théorie...

On considère une liste de  $n$  nombres, on peut la générer aléatoirement grâce à `L=[randint(1,n) for i in range(n)]` . Cette commande génère une liste de  $n$  entiers aléatoires, compris entre 1 et  $n$  (inclus).

```
[2]: def liste_al(n):  
      return [randint(1,n) for i in range(n)]  
  
L=liste_al(20)  
print(L)  
L.sort()  
print(L)
```

```
[6, 17, 16, 14, 2, 13, 13, 15, 3, 19, 16, 4, 11, 14, 14, 18, 6, 16, 3, 11]  
[2, 3, 3, 4, 6, 6, 11, 11, 13, 13, 14, 14, 14, 15, 16, 16, 16, 17, 18, 19]
```

Le travail de cette séance va consister à trier cette 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'intéressera plus particulièrement, de façon naturelle, à leur efficacité. Ainsi, on essayera de donner la complexité de ces algorithmes pour trier une liste de  $N$  éléments, et on donnera des évaluations numériques du temps nécessaire pour trier une liste.

Voici, 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 de coder par nous-même ces fonctions, afin de comprendre leur fonctionnement !

## 2 Trois algorithmes de tri

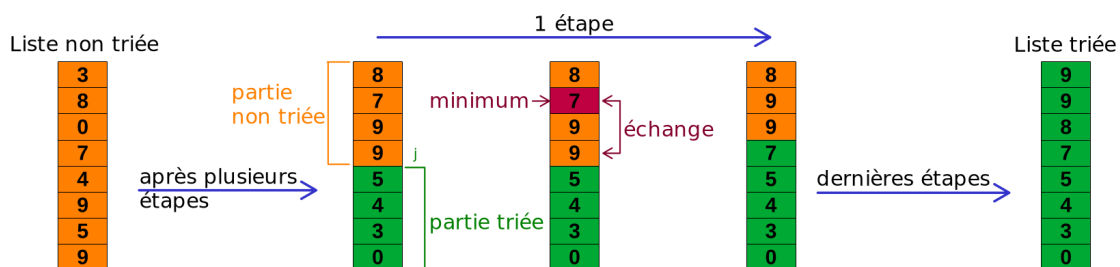
Dans le cadre du programme, nous allons étudier plusieurs algorithmes de tri, d'efficacités variées. Le site <https://professeurb.github.io/articles/tris/> présente des animations de ces différents algorithmes.

### 2.1 Un algorithme très naïf : le tri par sélection

La première idée qui peut nous venir à l'esprit pour trier une liste 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 été 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  $j$ ème position.

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



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

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

**Question 1 :** Écrire la fonction `minimum(L, j)` retournant la position du minimum de la liste  $L$ , à partir de l'indice  $j$  (inclus) :

```
[7]: 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)
      print(L)
      print(minimum(L,5))
```

```
[12, 2, 11, 9, 9, 8, 15, 12, 0, 13, 7, 12, 8, 12, 15]
8
```

**Question 2 :** Compléter alors la fonction `tri_naif(L)` retourner la liste issue de `L`, mais triée dans l'ordre croissant.

```
[15]: def tri_naif(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 :

```
[24]: L=liste_al(15)
      print(L)
      print(tri_naif(L))
```

```
[15, 13, 14, 2, 6, 3, 1, 3, 4, 13, 8, 10, 10, 6, 2]
[1, 2, 2, 3, 3, 4, 6, 6, 8, 10, 10, 13, 13, 14, 15]
```

*Remarque :* la portion de code `return L` est ici en réalité inutile ici, la liste `L` est modifiée chaque fois qu'on agit dessus.

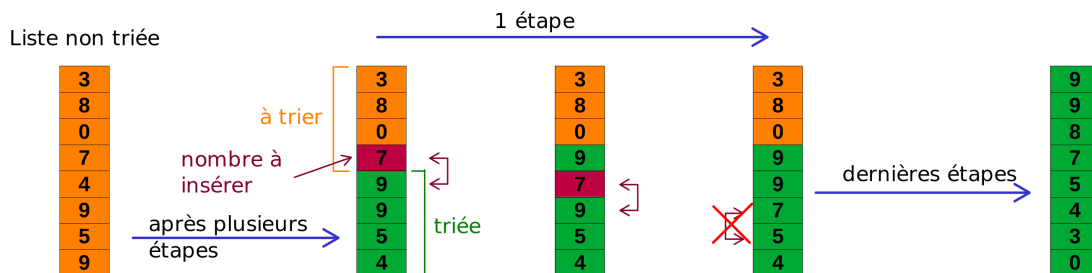
**Analyse de la complexité :** Cette algorithm est de complexité en  $\mathcal{O}(n^2)$ . En effet :

- on parcourt une fois la liste ->  $n$  "grosses" opérations

- pour chacune de ces grosses opérations, il faut chercher le 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

## 2.2 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 pourquoi nous allons commencer par une fonction d'insertion.

**Question 3 :** 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 `j`ème emplacement au "bon" emplacement :

```
[25]: 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 :

```
[44]: L=[1,2,3,4,5,6,7]+liste_al(10)
print(L)
print(insertion(L,7))
```

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

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

Écrivons alors la fonction de tri par insertion :

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

```
[26]: 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 à sa place
            insertion(L,i)
        return L
```

Vérifions son bon fonctionnement :

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

```
[3, 3, 13, 3, 15, 6, 8, 4, 6, 3, 15, 4, 11, 12, 0]
```

```
[0, 3, 3, 3, 3, 4, 4, 6, 6, 8, 11, 12, 13, 15, 15]
```

**Analyse de la complexité :** Cette algorithm 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 algorithm : en effet, les étapes sont en réalité très similaires (on parcourt la liste ->  $n$  opérations, et à chaque fois on effectue la comparaison d'un élément avec le  $n$  autres ->  $n \times n$  opérations). Cependant, cet algorithm est quand même un peu plus rapide :

```
[27]: Tnaif,Tinsertion,N=[],[],[]
        for n in range(50,1000,50):
            N.append(n)
            time1=time()
            for i in range (50): # on réalise 50 fois l'opération de triage, pour avoir
            ↪ une "valeur moyenne" cohérente
                L=liste_al(n)
                tri_naif(L)
            time2=time()
            for i in range (50):
                L=liste_al(n)
                tri_insertion(L)
            time3=time()
            Tnaif.append(time2-time1)
            Tinsertion.append(time3-time2)

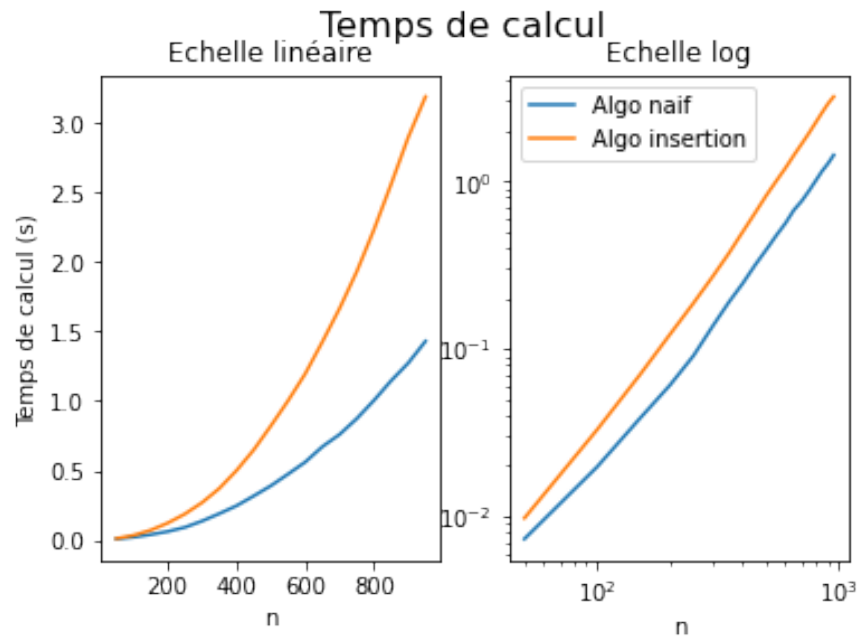
        fig, axs = plt.subplots(1, 2)
        axs[0].plot(N,Tnaif,label='Algo naif')
        axs[0].plot(N,Tinsertion,label='Algo insertion')
```

```

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

```



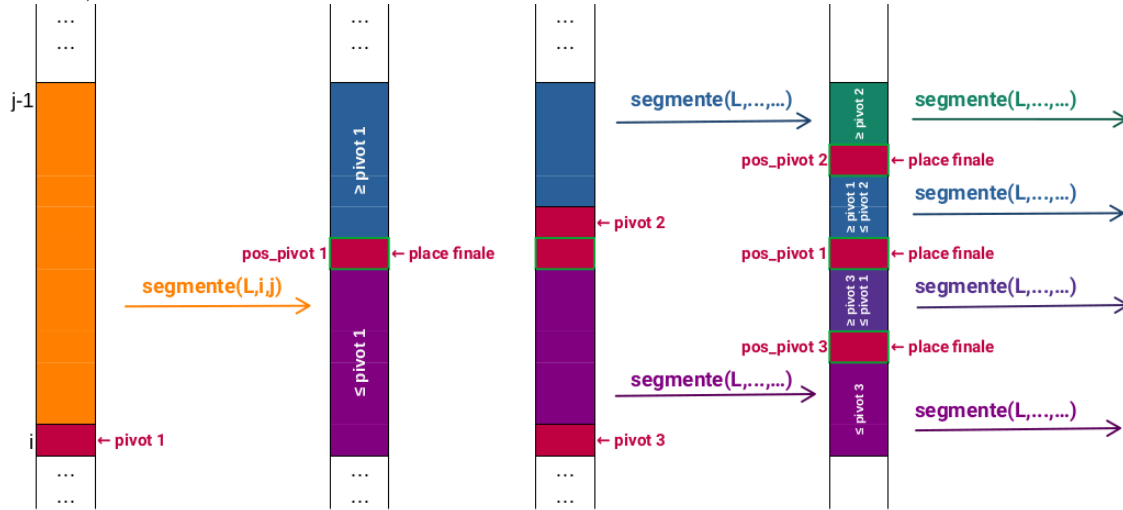
La pente en échelle log/log semble la même, ce qui est la signature de la même complexité. L'algorithme par insertion est cependant, de façon surprenante, un peu plus lent ici... Je pense que cette lenteur est imputable au fait que dans l'écriture de cet algorithme par insertion, on modifie en permanence la liste, opération coûteuse en temps.

### 2.3 Un algorithme rapide: le tri "rapide" (quicksort)

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.

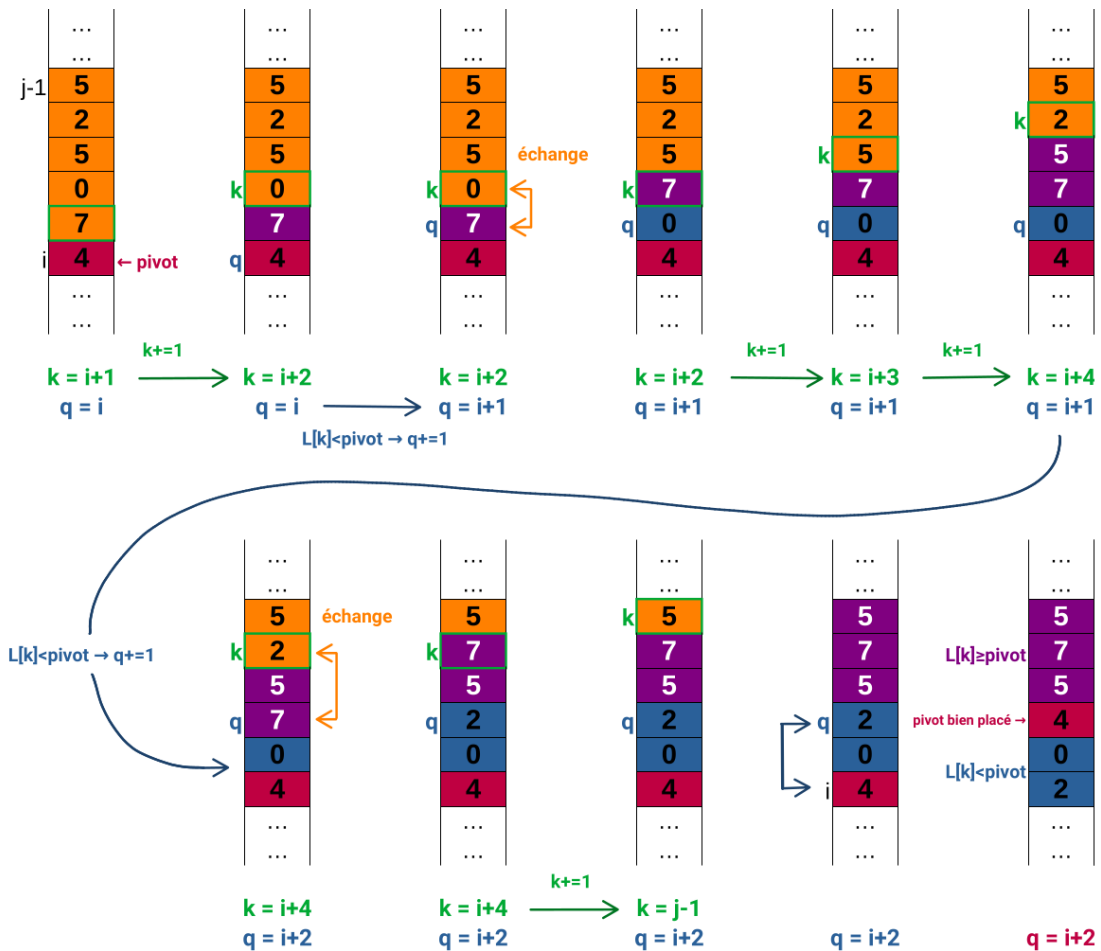
Voici ci-dessous le fonctionnement schématique de l'algorithme. Une partie de la liste, entre l'indice  $i$  (inclus) et l'indice  $j$  (exclus), est divisée (ou segmentée) en 3 parties, par le fonction `segmente` : cette fonction déplace un élément, nommé pivot, initialement situé en  $i$ , de sorte que les élé-

ments 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 précisément, 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 apparaît 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 5 :** 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).

```
[28] : 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 :

```

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

```

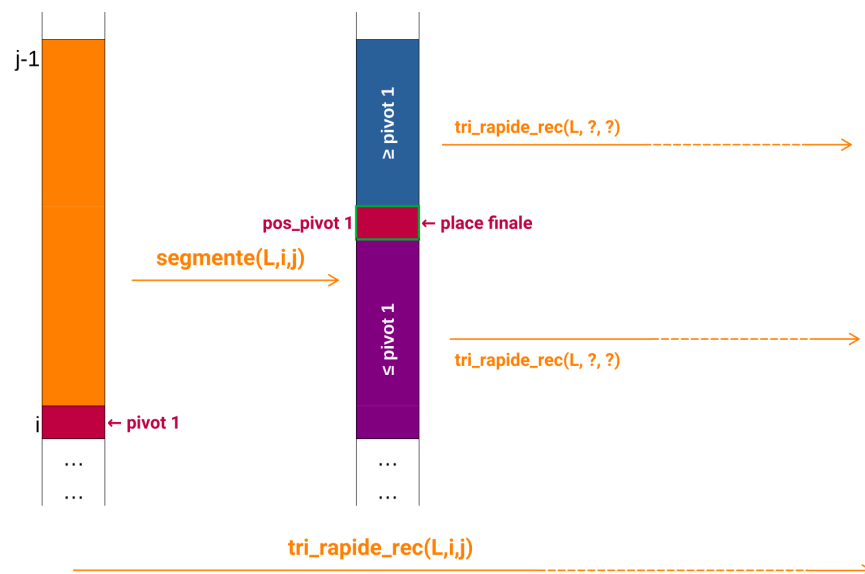
```

[0, 11, 1, 7, 8, 6, 8, 13, 7, 11, 6, 15, 13, 9, 14]
Position du pivot : 7 Valeur du pivot : 8
[0, 11, 1, 7, 6, 6, 7, 8, 8, 11, 13, 15, 13, 9, 14]

```

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

**Question 6 :** Écrire la fonction `tri_rapide_rec(L,i,j)` qui tri la portion située entre `i` et `j` (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”.

```
[29]: 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) :

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

Vérifions :

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

```
[9, 0, 2, 10, 1, 0, 9, 3, 13, 7, 8, 3, 15, 7, 0]
[0, 0, 0, 1, 2, 3, 3, 7, 7, 8, 9, 9, 10, 13, 15]
```

Petit test de rapidité :

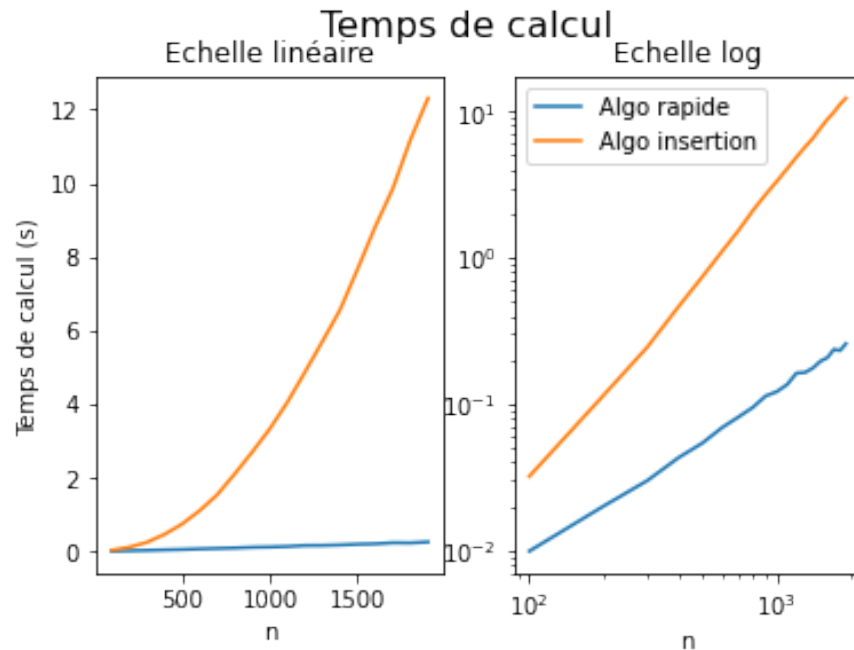
```
[32]: Trapide,Tinsertion,N=[],[],[]
        for n in range(100,2000,100):
            N.append(n)
            time1=time()
            for i in range(50):
                L=liste_al(n)
                tri_rapide(L)
            time2=time()
            for i in range(50):
                L=liste_al(n)
                tri_insertion(L)
            time3=time()
            Trapide.append(time2-time1)
            Tinsertion.append(time3-time2)

        fig, axs = plt.subplots(1, 2)
        axs[0].plot(N,Trapide,label='Algo rapide')
        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,Trapide,label='Algo rapide')
axs[1].loglog(N,Tinsertion,label='Algo insertion')
axs[1].set_title('Echelle log')
axs[1].set_xlabel('n')
plt.legend()
plt.show()

```



Effectivement, le tri rapide porte bien son nom ! Il est effectivement 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.

**Pour aller plus loin :**

- On peut écrire, de façon plus plus élégante :

```

[34]: 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  $\mathcal{O}(n \log(n))$  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.

```
[35]: 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
              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 + 20 < 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_insertion(L[i:j])
    return L
```

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

```
[4, 4, 3, 10, 6, 14, 24, 18, 23, 13, 19, 15, 17, 1, 29, 4, 30, 23, 22, 25, 22,
19, 30, 17, 2, 4, 12, 2, 23, 23]
[1, 2, 2, 3, 4, 4, 4, 4, 6, 10, 12, 13, 14, 15, 17, 17, 18, 19, 19, 22, 22, 23,
23, 23, 23, 24, 25, 29, 30, 30]
```

```
[36]: Trapide3,Trapide,N=[], [], []
      for n in range(100,2000,100):
          N.append(n)
          time1=time()
```

```

for i in range(50):
    L=liste_al(n)
    tri_rapide2(L)
time2=time()
for i in range(50):
    L=liste_al(n)
    tri_rapide3(L)
time3=time()
Trapide.append(time2-time1)
Trapide3.append(time3-time2)

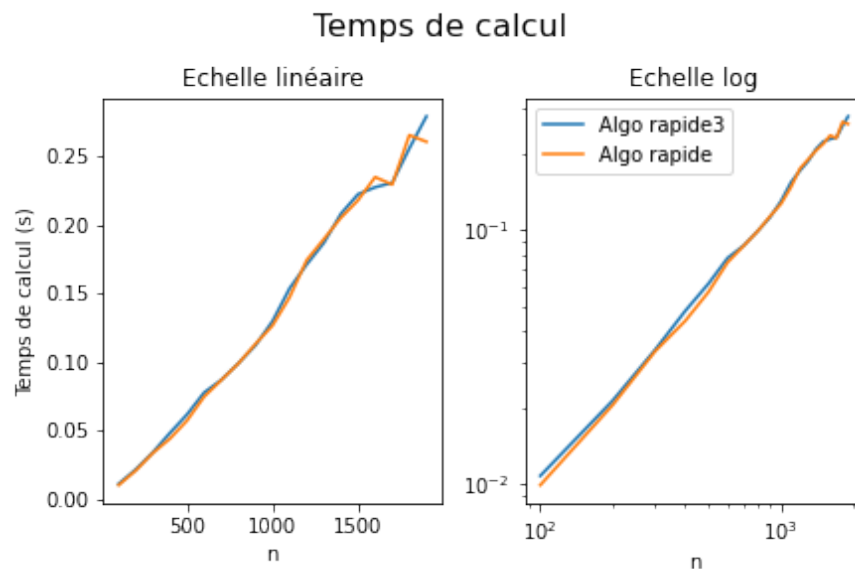
```

```

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 :

```

[37]: Trapide,Tsort,N=[], [], []
for n in range(100,2000,100):

```

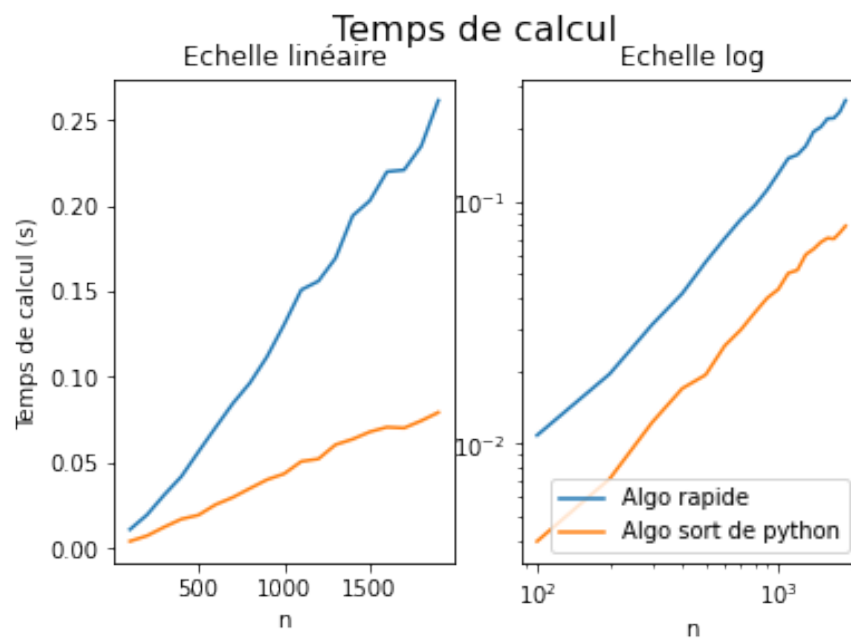
```

N.append(n)
time1=time()
for i in range(50):
    L=liste_al(n)
    tri_rapide(L)
time2=time()
for i in range(50):
    L=liste_al(n)
    L.sort()
time3=time()
Trapide.append(time2-time1)
Tsort.append(time3-time2)

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*.

## 2.4 Résumé

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

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

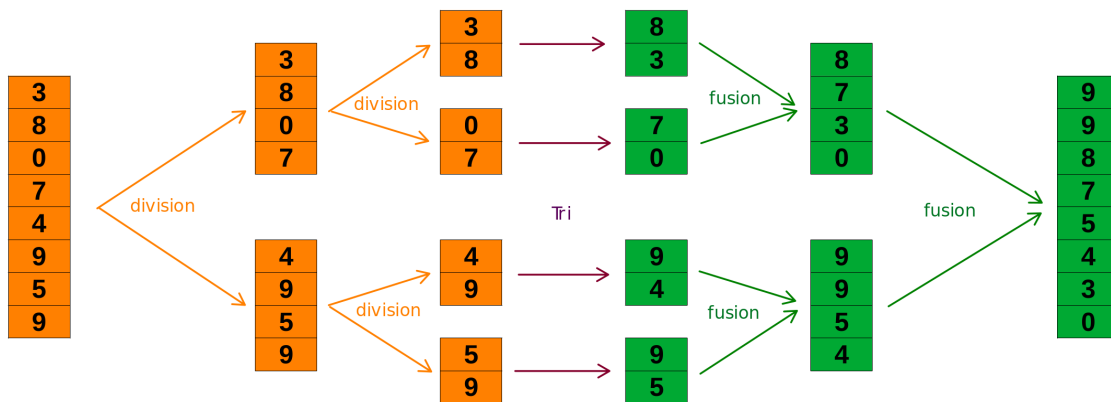
## 3 Pour aller plus loin

### 3.1 Un dernier algorithme de tri : l'algorithme de tri fusion

La stratégie est toujours basée sur le "Division 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.

```
[38]: 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[i1+i2:]=L1[i1:]
    # ou L.extend(L1[i1:])
    # Sinon on rajoute le reste de L2
    else:
        L[i1+i2:]=L2[i2:]
    # ou L.extend(L2[i2:])
    return L

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

```

La complexité est toujours en  $\mathcal{O}(n \log n)$ .

```

[39]: Trapide,Tfusion,N=[],[],[]
for n in range(50,1000,50):
    N.append(n)
    time1=time()
    for i in range(50):
        L=liste_al(n)
        tri_rapide(L)
    time2=time()
    for i in range(50):
        L=liste_al(n)
        tri_fusion(L)
    time3=time()
    Trapide.append(time2-time1)
    Tfusion.append(time3-time2)

fig, axs = plt.subplots(1, 2, constrained_layout=True)
axs[0].plot(N,Trapide,label='Algo rapide')
axs[0].plot(N,Tfusion,label='Algo fusion')
axs[0].set_title('Echelle linéaire')
axs[0].set_xlabel('n')

```

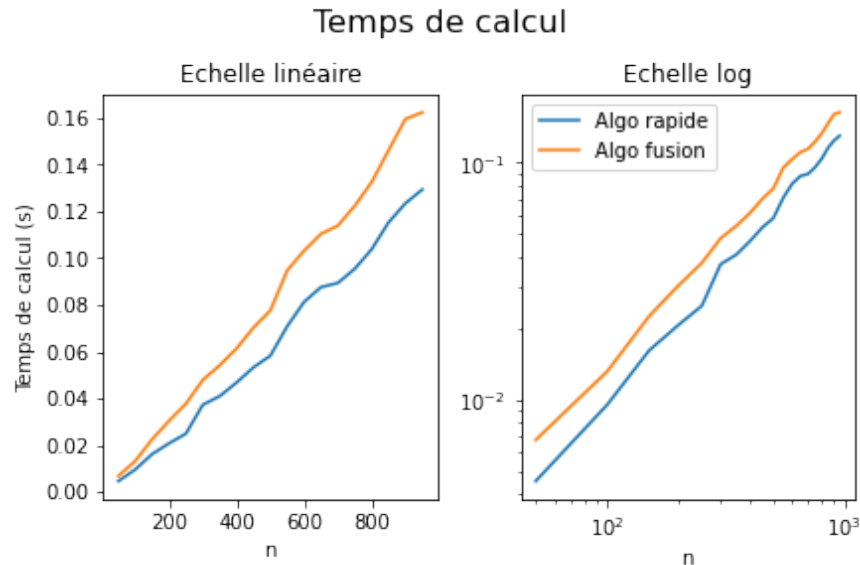


```

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

```



Le tri par fusion est plus lent que le tri rapide, mais semble posséder la même complexité !

### 3.2 Utilisation d'un algorithme de tri : détermination de la médiane

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. La première méthode pour trouver la médiane consiste à trier la liste, puis à prendre l'élément central.

**Question 8:** Ecrire une fonction `f_médiane_normale(L)` qui détermine la médiane d'une liste `L` par réalisation du tri rapide sur `L` puis par détermination de son élément central.

Il existe une méthode moins coûteuse 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.

**Question 9:** Après avoir réfléchi et trouver comment programmer cet algorithme, proposer une fonction `f_médiane_rapide(L)` permettant de le réaliser. On pourra définir une fonction intermédiaire si besoin !

**Question 10:** Comparer les temps d'exécution de ces deux algorithmes.

Correction question 8 :

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

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

[1, 9, 1, 10, 10, 10, 16, 5, 19, 19, 5, 11, 19, 12, 20, 14, 16, 15, 17, 17]

Médiane : 14

Liste triée : [1, 1, 5, 5, 9, 10, 10, 10, 11, 12, 14, 15, 16, 16, 17, 17, 19, 19, 19, 20]

Correction question 9 : 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.

```
[41]: 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)]
```

```
[65]: L=liste_al(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)
```

[17, 11, 17, 1, 15, 8, 9, 1, 11, 14, 3, 12, 15, 8, 16, 6, 0, 20, 5, 11]

Médiane par méthode rapide: 11

Médiane par méthode normale: 11

Liste triée : [0, 1, 1, 3, 5, 6, 8, 8, 9, 11, 11, 11, 12, 14, 15, 15, 16, 17, 17, 20]

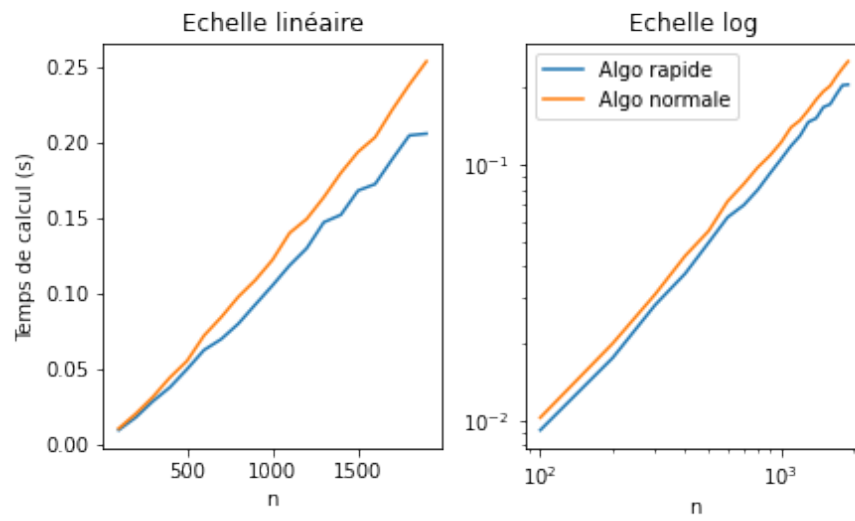
Correction question 10 :

```
[42]: Trapide,Tnormale,N=[],[],[]
for n in range(100,2000,100):
    N.append(n)
    time1=time()
    for i in range(50):
        L=liste_al(n)
        f_mediane_rapide(L)
    time2=time()
    for i in range(50):
        L=liste_al(n)
        f_mediane_normale(L)
    time3=time()
    Trapide.append(time2-time1)
    Tnormale.append(time3-time2)

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()
```

## Temps de calcul



On gagne (très peu, mais on gagne !)