Tri rapide

Hoare, C. A. R. (1961). "Algorithm 64: Quicksort". Comm. ACM. 4 (7): 321

Principe

Le tri rapide est un algorithme récursif consistant à

- choisir une valeur pivot parmi les éléments du tableau
- partitionner le tableau en: [<= pivot] pivot [>= pivot]
- appeler récursivement le tri rapide sur les 2 partitions

Partition

Entrée:

Le sous tableau `T[premier:dernier]` dont le dernier élément sert de pivot

Sortie:

L'indice `i` du pivot dans le tableau partitionné tel que

- `T[premier:i]` contient les éléments `<= pivot`
- `T[i]` contient le pivot
- `T[i+1:dernier]` contient les éléments `>= pivot`

Algorithme: deux parcours simultanés de tableau

- croissant (`i`), s'arrête quand `T[i] >= pivot`
- décroissant (`j`), s'arrête quand `T[j] <= pivot`

à chaque arrêt de `i` et `j`, on permute `T[i]` et `T[j]`, puis on continue

Parcours croissant (i)

In [1]:
```
def i_suivant(T,i,pivot):
    while i < pivot and T[i] < T[pivot]:
        i += 1
    return i
```

In [2]:
```
T = [ 7, 6, 2, 1, 3, 5, 8, 4 ]; pivot = len(T)-1

i = i_suivant(T,0,pivot)
print("T [",i,"] =",T[i],">=",T[pivot])
```

```
T [ 0 ] = 7 >= 4
```

In [3]:
```
i = i_suivant(T,i+1,pivot)
print("T [",i,"] =",T[i],">=",T[pivot])
```

```
T [ 1 ] = 6 >= 4
```

In [4]:
```
i = i_suivant(T,i+1,pivot)
print("T [",i,"] =",T[i],">=",T[pivot])
```

```
T [ 5 ] = 5 >= 4
```

Parcours décroissant (j)

In [5]:
```
def j_precedent(T,j,pivot):
    while j >= 0 and T[j] > T[pivot]:
        j -= 1
    return j
```

In [6]:
```
T = [ 7, 6, 2, 1, 3, 5, 8, 4 ]; pivot = len(T)-1

j = j_precedent(T,pivot-1,pivot)
print("T [",j,"] =",T[j],"<=",T[pivot])
```

```
T [ 4 ] = 3 <= 4
```

In [7]:
```
j = j_precedent(T,j-1,pivot)
print("T [",j,"] =",T[j],"<=",T[pivot])
```

```
T [ 3 ] = 1 <= 4
```

In [8]:
```
j = j_precedent(T,j-1,pivot)
print("T [",j,"] =",T[j],"<=",T[pivot])
```

```
T [ 2 ] = 2 <= 4
```

Partition complète

- effectuer les deux parcours
- échanger à chaque double arrêt
- s'arrêter quand les parcours se croisent.
- placer le pivot à la bonne place

In [9]:

```
def partition(T,premier,dernier):
    pivot = dernier-1
    i = premier; j = pivot-1

    while True:
        i = i_suivant(T,i,pivot)
        j = j_precedent(T,j,pivot)
        if j < i:
            break
        T[i],T[j] = T[j],T[i]

    T[i],T[pivot] = T[pivot],T[i]

    return i
```

Appliquons cette fonction sur un exemple

In [10]:

```
T = [ 7, 6, 2, 1, 3, 5, 8, 4 ]
p1 = partition(T,0,len(T))
print(T[0:p1],T[p1],T[p1+1:len(T)])
```

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

In [11]:

```
p2 = partition(T,0,p1)
print(T[0:p2],T[p2],T[p2+1:p1])
```

```
[1] 2 [3]
```

In [12]:

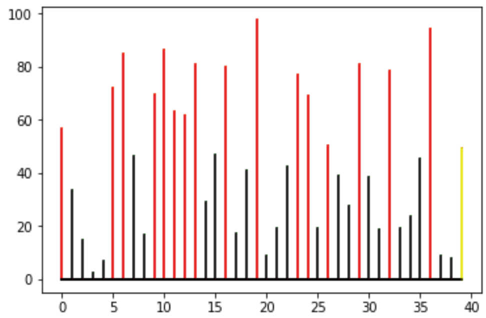
```
p3 = partition(T,p1+1,len(T))
print(T[p1+1:p3],T[p3],T[p3+1:len(T)])
```

```
[5] 6 [8, 7]
```

Visualisons la partition

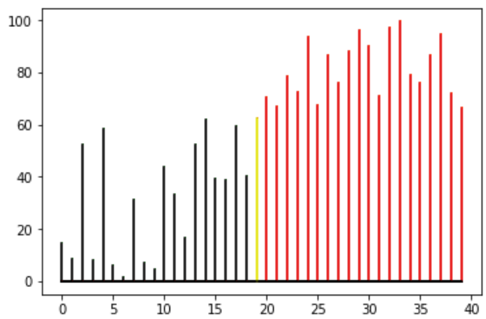
In [48]:
```
import numpy as np
import include.helpers as asd1

T = np.random.uniform(0,100,40)
T[len(T)-1] = np.average(T) + 5
asd1.affiche_partition(T,0,len(T),len(T)-1)
```



In [14]:
```
i = partition(T,0,len(T))

asd1.affiche_partition(T,0,len(T),i)
```



Algorithme récursif

Cas trivial

un tableau de 0 ou 1 élément est déjà trié.

Cas général

- Choisir un pivot
- Partitionner autour de ce pivot
- trier récursivement les deux côtés de la partition

In [16]:

```python
def tri_rapide_rec(T,premier,dernier):
    if premier < dernier-1:           # >= 2 éléments
        pivot = dernier - 1           # choix du pivot
        T[pivot],T[dernier-1] = T[dernier-1],T[pivot]

        pivot = partition(T,premier,dernier)
        afficher_partition(T,premier,pivot,dernier)

        tri_rapide_rec(T,premier,pivot)
        tri_rapide_rec(T,pivot+1,dernier)
```

In [17]:

```python
T = [ 7, 6, 2, 1, 3, 5, 8, 4 ]
tri_rapide_rec(T,0,len(T))
```

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

En résumé

In [18]:

```
def partition(T,premier,dernier,
              comparer = asd1.plus_petit):

    pivot = dernier-1; i = premier; j = pivot-1

    while True:
        while i < pivot    and comparer(T[i],T[pivot]): i += 1
        while j >= premier and comparer(T[pivot],T[j]): j -= 1
        if j < i: break
        asd1.echanger(T,i,j)
        i += 1; j -= 1

    asd1.echanger(T,i,pivot)
    return i
```

In [19]:

```
def tri_rapide_rec(T,premier,dernier,
                   comparer = asd1.plus_petit):

    if premier < dernier-1:
        pivot = choix_du_pivot(T,premier,dernier);
        asd1.echanger(T,pivot,dernier-1)

        pivot = partition(T,premier,dernier,comparer)

        tri_rapide_rec(T,premier,pivot,comparer)
        tri_rapide_rec(T,pivot+1,dernier,comparer)
```

In [20]:

```
def choix_du_pivot(T,premier,dernier):
    return dernier-1
```

In [21]:

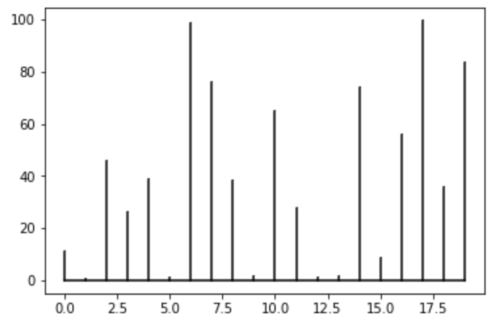
```
def tri_rapide(T,comparer = asd1.plus_petit):
    tri_rapide_rec(T,0,len(T),comparer)
```

Visualisation

Observons les premières partition du tri de 20 entiers aléatoires entre 0 et 100

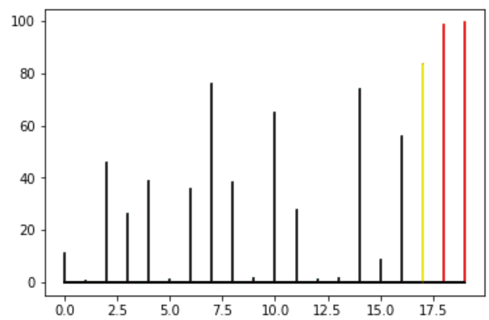
In [57]:

```
import matplotlib.pyplot as plt
T = np.random.uniform(0,100,20)
plt.stem(T,markerfmt=',',linefmt='black',basefmt='black'); plt.
show()
```

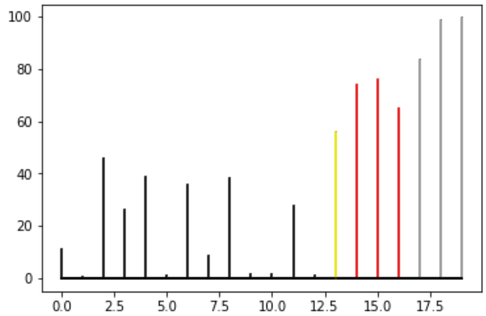


In [58]:

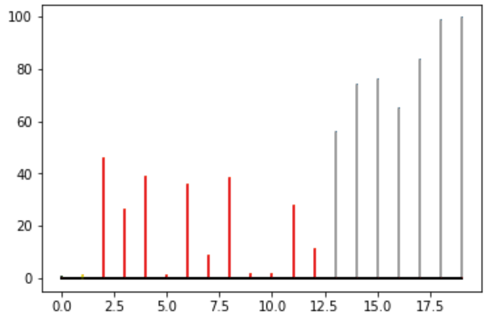
```
p1 = partition(T,0,len(T))
asd1.affiche_partition(T,0,len(T),p1)
```



In [59]:
```
p2 = partition(T,0,p1)
asd1.affiche_partition(T,0,p1,p2)
```



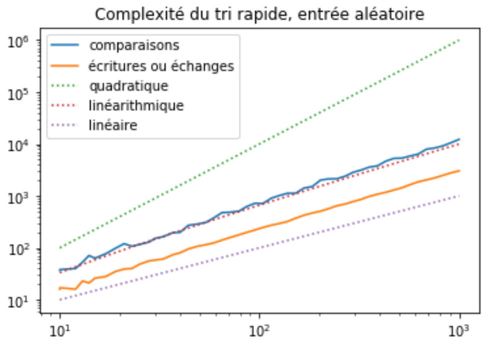
In [60]:
```
p3 = partition(T,0,p2)
asd1.affiche_partition(T,0,p2,p3)
```



Complexité

Evaluons la complexité avec une entrée aléatoire

In [26]:
```
asd1.evalue_complexite(tri_rapide, asd1.tableau_aleatoire,
                       "tri rapide, entrée aléatoire")
```

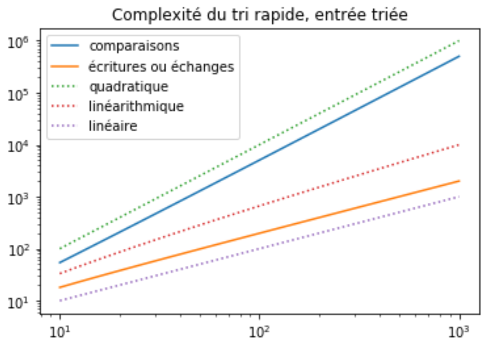


N	Comp.	Ecr.
10	37	16
19	98	35
37	197	74
71	483	157
138	1121	321
268	2381	696
517	5343	1431
1000	12264	3037

Nb comparaisons: $\Theta(n \log n)$. Nb échanges < Nb comparaisons

Voyons maintenant ce qui se passe avec une entrée triée

In [27]:
```
asd1.evalue_complexite(tri_rapide, asd1.tableau_trie,
                       "tri rapide, entrée triée")
```



N	Comp.	Ecr.
10	54	18
19	189	36
37	702	72
71	2555	140
138	9590	274
268	36045	534
517	133902	1032
1000	500499	1998

La complexité est maintenant quadratique en $\Theta(n^2)$.

Observons ce qui se passe en détail

In [28]:
```
T = [ 1, 2, 3, 4, 5, 6, 7, 8]
p1 = partition(T,0,len(T)); print(T[0:p1],T[p1],T[p1+1:len(T)])
```

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

In [29]:
```
p2 = partition(T,0,p1); print(T[0:p2],T[p2],T[p2+1:p1])
```

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

In [30]:
```
p3 = partition(T,0,p2); print(T[0:p3],T[p3],T[p3+1:p2])
```

```
[1, 2, 3, 4, 5] 6 []
```

In [31]:
```
p4 = partition(T,0,p3); print(T[0:p4],T[p4],T[p4+1:p3])
```

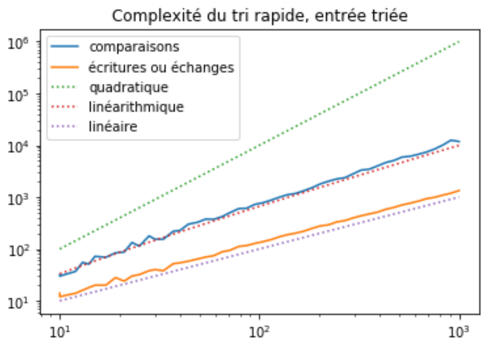
```
[1, 2, 3, 4] 5 []
```

La stratégie de choix du pivot et mauvaise.

Choix du pivot aléatoire

In [32]:
```
def choix_du_pivot(T,premier,dernier):
    return np.random.randint(premier,dernier)

asd1.evalue_complexite(tri_rapide, asd1.tableau_trie,
                       "tri rapide, entrée triée")
```



N	Comp.	Ecr.
10	32	14
19	84	28
37	219	52
71	499	94
138	1110	188
268	2362	352
517	5895	704
1000	11811	1336

Avec un choix aléatoire du pivot, la complexité est de nouveau linéarithmique.

Analyse théorique

Soit C_n le nombre de comparaisons nécessaires pour le tri rapide de n éléments.

$$C_n = (n+1) + C_k + C_{n-1-k}$$

avec

- $n+1$ le nombre de comparaisons pour la partition
- k le nombre d'éléments à gauche du pivot après partition
- $n-k-1$ le nombre d'éléments à droite du pivot après partition

Avec un choix de pivot aléatoire, toutes les valeurs de k entre 0 et $n-1$ sont équiprobables, de probabilité $\frac{1}{n}$.

C_n vont donc en moyenne

$$C_n = (n+1) + \sum_{k=0}^{n-1} \frac{1}{n}(C_k + C_{n-1-k})$$

et en notant que $\sum_{k=0}^{n-1} C_{n-1-k} = \sum_{u=0}^{n-1} C_u$ pour $u = n-1-k$, on obtient

$$C_n = (n+1) + \frac{2}{n} \cdot \sum_{k=0}^{n-1} C_k$$

Pour simplifier cette équation, notons que

$$n \cdot C_n = n^2 + n + 2 \cdot \sum_{k=0}^{n-1} C_k$$

$$(n-1) \cdot C_{n-1} = n^2 - n + 2 \cdot \sum_{k=0}^{n-2} C_k$$

En prenant la différence entre les deux, on obtient

$$n \cdot C_n - (n-1) \cdot C_{n-1} = 2 \cdot n + 2 \cdot C_{n-1}$$

En regroupant les termes et divisant par $n(n+1)$, on trouve

$$\frac{C_n}{n+1} = \frac{C_{n-1}}{n} + \frac{2}{n+1}$$

en tenant compte de ce que $C_0 = C_1 = 0$ pour des tableaux de 0 ou 1 éléments, on résoud cet équation récursive ce qui donne

$$C_n = 2 \cdot (n+1) \cdot \sum_{k=3}^{n+1} \frac{1}{k}$$

en approximant cette somme par une intégrale, on trouve

$$C_n \approx 2 \cdot (n+1) \cdot \ln n$$

en logarithme népérien, ce qui donne

$$C_n \approx 1.39 \cdot n \cdot \log n$$

en logarithme binaire.

Choix du pivot

Un choix aléatoire du pivot n'est évidemment pas optimal.

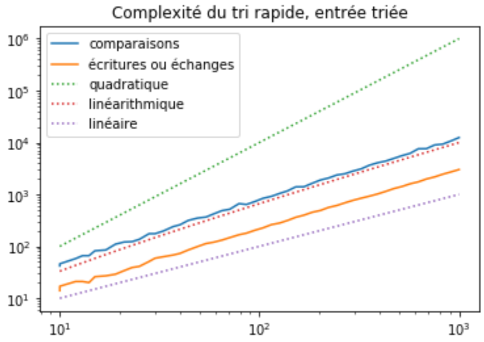
Le **choix optimal** serait de prendre la **valeur médiane**, qui partitionne en deux parts égales. Mais trouver cette médiane est trop cher

Un bon compromis consiste à prendre la médiane de 3 éléments

In [33]:
```
def index_median(T,i1,i2,i3):
    if asd1.plus_petit(T[i1],T[i2]):
        if asd1.plus_petit(T[i2],T[i3]):    return i2
        elif asd1.plus_petit(T[i1],T[i3]):  return i3
        else:                               return i1
    else:
        if asd1.plus_petit(T[i1],T[i3]):    return i1
        elif asd1.plus_petit(T[i2],T[i3]):  return i3
        else:                               return i2
```

In [34]:
```
def choix_du_pivot(T,premier,dernier):
    milieu = premier+(dernier-premier)//2
    return index_median(T,premier,dernier-1,milieu)

asd1.evalue_complexite(tri_rapide, asd1.tableau_aleatoire,
                       "tri rapide, entrée triée")
```



N	Comp.	Ecr.
10	42	14
19	110	29
37	240	68
71	517	147
138	1164	314
268	2498	688
517	5565	1417
1000	12432	3010

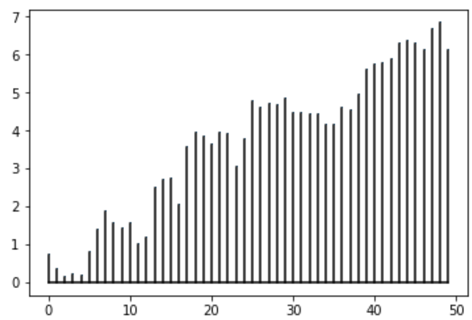
Stabilité

Le tri rapide n'est **pas stable**.

L'opération de partition déplace les éléments selon leur comparaison avec le pivot et rien ne garantit que deux éléments égaux restent ordonnés pareillement.

In [35]:
```
asd1.test_stabilite(tri_rapide)
```

```
Le tri n'est pas stable
```



Traitement des valeurs égales au pivot

Que se passe-t-il si le tableau contient des valeurs répétées ?

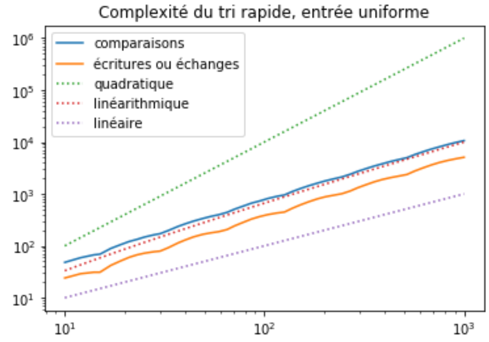
In [36]:
```
def tableau_uniforme(n):
    return [1]*n

T = tableau_uniforme(10)
p1 = partition(T,0,len(T)); print(T[0:p1],T[p1],T[p1+1:len(T)])
```

```
[1, 1, 1, 1, 1] 1 [1, 1, 1, 1]
```

Les valeurs égales au pivot sont réparties dans les 2 parties

In [37]:
```
asd1.evalue_complexite(tri_rapide, tableau_uniforme,
                       "tri rapide, entrée uniforme")
```



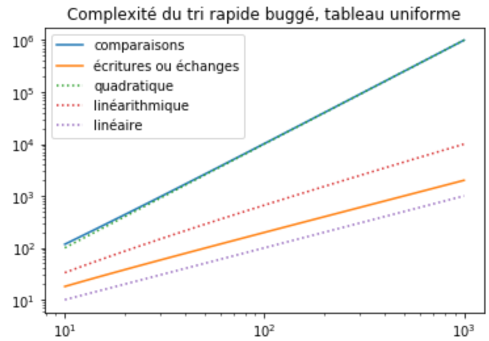
N	Comp.	Ecr.
10	48	24
19	105	51
37	237	114
71	509	243
138	1110	529
268	2376	1131
517	4973	2362
1000	10607	5071

Le tri rapide reste linéarithmique en moyenne.

Mais attention ... on a fait le choix surprenant d'échanger les valeurs égales au pivot lors de la partition

i.e., on teste `T[i] < T[pivot]` et pas `T[i] <= T[pivot]` en cherchant l'indice suivant. Sinon...

In [38]:
```
def tri_rapide_bugge(T): tri_rapide(T,asd1.plus_petit_ou_egal)
asd1.evalue_complexite(tri_rapide_bugge, tableau_uniforme,
                       "tri rapide buggé, tableau uniforme")
```



N	Comp.	Ecr.
10	117	18
19	396	36
37	1440	72
71	5180	140
138	19317	274
268	72357	534
517	268320	1032
1000	1001997	1998

Voyons ce qui se passe en détail

In [39]:
```
T = tableau_uniforme(10)
p1 = partition(T,0,len(T),asd1.plus_petit_ou_egal)
print(T[0:p1],T[p1],T[p1+1:len(T)])
```

```
[1, 1, 1, 1, 1, 1, 1, 1, 1] 1 []
```

In [40]:
```
p2 = partition(T,0,p1,asd1.plus_petit_ou_egal)
print(T[0:p2],T[p2],T[p2+1:p1])
```

```
[1, 1, 1, 1, 1, 1, 1, 1] 1 []
```

In [41]:
```
p3 = partition(T,0,p2,asd1.plus_petit_ou_egal)
print(T[0:p3],T[p3],T[p3+1:p2])
```

```
[1, 1, 1, 1, 1, 1, 1] 1 []
```

La plupart des mises en oeuvres de `qsort` de la librairie C standard avaient ce bug jusqu'en 1991...

Dé-récursification

Même si un bon choix de pivot le rend extrêmement improbable, le pire cas ne pose pas qu'un problème de complexité.

Si le choix du pivot ne diminue que de 1 élément la taille de la partition, le **profondeur de récursion** est de $\Theta(n)$.

Cela peut entraîner un **débordement de la pile** de récursion.

Pour éviter ce problème, on va remplacer un des 2 appels récursifs par une **boucle**

Affichons les appels effectués par le tri doublement récursif.

In [42]:
```
def tri_rapide_recursif(T,premier,dernier):
    if premier < dernier-1:
        print("appel(T,{0},{1})".format(premier,dernier))
        pivot = partition(T,premier,dernier)
        afficher_partition(T,premier,pivot,dernier)
        tri_rapide_recursif(T,premier,pivot)
        tri_rapide_recursif(T,pivot+1,dernier)
```

In [43]:
```
T = [ 7, 6, 4, 3, 2, 1, 8, 5 ]
tri_rapide_recursif(T,0,len(T))
```

```
appel(T,0,8)
 [1, 2, 4, 3] 5 [7, 8, 6]
appel(T,0,4)
 [1, 2] 3 [4]
appel(T,0,2)
 [1] 2 []
appel(T,5,8)
                [] 6 [8, 7]
appel(T,6,8)
                   [] 7 [8]
```

Remplaçons le **deuxième appel** par une boucle `while`

In [44]:
```
def tri_rapide_semi_recursif(T,premier,dernier):
    if premier < dernier-1:
        print("appel(T,{0},{1})".format(premier,dernier))

    while premier < dernier-1:      # while remplace if

        pivot = partition(T,premier,dernier)
        afficher_partition(T,premier,pivot,dernier)
        tri_rapide_semi_recursif(T,premier,pivot)
        premier = pivot+1;          # remplace appel récursif
```

In [45]:
```
T = [ 7, 6, 4, 3, 2, 1, 8, 5 ]
tri_rapide_semi_recursif(T,0,len(T))
```

```
appel(T,0,8)
 [1, 2, 4, 3] 5 [7, 8, 6]
appel(T,0,4)
 [1, 2] 3 [4]
appel(T,0,2)
 [1] 2 []
                [] 6 [8, 7]
                   [] 7 [8]
```

Mieux, choisissons **l'intervalle le plus court** pour l'appel récursif

In [46]:
```
def tri_rapide_recursion_minimale(T,premier,dernier):
    if premier < dernier-1:
        print("appel(T,{0},{1})".format(premier,dernier))

    while premier < dernier-1:
        pivot = partition(T,premier,dernier)
        afficher_partition(T,premier,pivot,dernier)
        if pivot - premier < dernier - (pivot+1):
            tri_rapide_recursion_minimale(T,premier,pivot)
            premier = pivot+1;
        else:
            tri_rapide_recursion_minimale(T,pivot+1,dernier)
            dernier = pivot
```

In [47]:
```
T = [ 7, 6, 4, 3, 2, 1, 8, 5 ]
tri_rapide_recursion_minimale(T,0,len(T))
```

```
appel(T,0,8)
 [1, 2, 4, 3] 5 [7, 8, 6]
appel(T,5,8)
                [] 6 [8, 7]
                   [] 7 [8]
 [1, 2] 3 [4]
 [1] 2 []
```

En remplaçant l'appel récursif pour l'intervalle le plus long par une boucle,

la taille de l'intervalle pour l'appel récursif est au moins divisée par 2 à chaque appel

la **profondeur de récursion** est au plus de $\mathcal{O}(\log(n))$

Complexité spatiale

La **partition** s'effectue en place. Elle n'utilise qu'un nombre constant de variables auxilliaires

- les compteurs `i` et `j`
- une variable temporaire pour les échanges

La complexité spatiale de la partition est donc $\Theta(1)$.

Mais ...

Le tri rapide est un **algorithme récursif**, et chaque appel récursif nécessite de la mémoire pour

- la pile d'appels
- les variables locales

La complexité spatiale du tri rapide est donc **proportionelle à la profondeur de récursion maximale**.

- $\Theta(\log(n))$ en moyenne et $\Theta(n)$ au pire pour la version doublement récursive
- $\Theta(\log(n))$ au pire pour la version remplaçant une récursion par une boucle

Conclusion

Le tri rapide

- n'est pas stable
- a une complexité temporelle moyenne en $\Theta(n\log(n))$
- a une complexité temporelle en $\Theta(n^2)$ dans le pire des cas
  - peu probable avec un bon choix de pivot
  - très probable (entrée triée) avec un mauvais choix
- a une complexité spatiale en $\Theta(\log(n))$ quand il est bien mis en oeuvre
- est plus rapide que le tri fusion en pratique (moins d'écritures, meilleure utilisation de la mémoire cache)

heig-vd

[ASD1 Notebooks on GitHub.io (https://ocuisenaire.github.io/ASD1-notebooks/)](https://ocuisenaire.github.io/ASD1-notebooks/)