

Algorithmes de tri

Informatique pour tous



Question

Comment savoir si un élément e appartient à une liste L ?

Question

Comment savoir si un élément e appartient à une liste L ?

```
def contient(L, e):  
    for i in range(len(L)):  
        if L[i] == e:  
            return True  
    return False
```

Question

Comment savoir si un élément e appartient à une liste L ?

```
def contient(L, e):  
    for i in range(len(L)):  
        if L[i] == e:  
            return True  
    return False
```

Complexité dans le pire cas : $O(n)$, où $n = \text{len}(L)$.

Recherche dichotomique

Si L est **triée**, on peut savoir si e est dans L plus rapidement, en comparant e avec le **milieu** m de L :

- Si $e == m$, on a trouvé notre élément.
- Si $e > m$, il faut chercher e dans la partie droite de L
- Si $e < m$, il faut chercher e dans la partie gauche de L

Recherche dichotomique : exemple

On veut savoir si 14 appartient à la liste :

$$L = [-2, 1, 2, 4, 6, 7, 8, 9, 11, 12, 14, 15, 18, 22, 54]$$

Recherche dichotomique : exemple

On veut savoir si 14 appartient à la liste :

[-2, 1, 2, 4, 6, 7, 8, 9, 11, 12, 14, 15, 18, 22, 54]

$$9 < 14$$

Recherche dichotomique : exemple

On veut savoir si 14 appartient à la liste :

`[-2, 1, 2, 4, 6, 7, 8, 9, 11, 12, 14, 15, 18, 22, 54]`

$$9 < 14$$

Recherche dichotomique : exemple

On veut savoir si 14 appartient à la liste :

[-2, 1, 2, 4, 6, 7, 8, 9, 11, 12, 14, 15, 18, 22, 54]

15 > 14

Recherche dichotomique : exemple

On veut savoir si 14 appartient à la liste :

$[-2, 1, 2, 4, 6, 7, 8, 9, \boxed{11, \underline{12}, 14}, 15, 18, 22, 54]$

$$12 < 14$$

Recherche dichotomique : exemple

On veut savoir si 14 appartient à la liste :

[-2, 1, 2, 4, 6, 7, 8, 9, 11, 12, 14, 15, 18, 22, 54]

14 trouvé !

Recherche dichotomique : version itérative

```
def contient_dichotomie(L, e):  
    debut = 0 # indice de début  
    fin = len(L) # indice de fin exclu  
    while debut < fin:  
        milieu = (debut + fin) // 2  
        if L[milieu] == e:  
            return True  
        elif L[milieu] < e: # il faut chercher à droite  
            debut = milieu + 1  
        else: # il faut chercher à gauche  
            fin = milieu  
    return False
```

Recherche dichotomique : version récursive

Recherche dichotomique : version récursive

`dicho(L, e, i, j)` cherche `e` dans `L` entre les indices `i` et `j - 1` :

Recherche dichotomique : version récursive

dicho(L, e, i, j) cherche e dans L entre les indices i et j - 1 :

```
def dicho(L, e, i, j):  
    if i >= j: return False  
    m = (i + j) // 2 # milieu de i et j  
    if L[m] == e: return True  
    if e < L[m]: # il faut chercher à gauche de m  
        return dicho(L, e, i, m)  
    else: # il faut chercher à droite de m  
        return dicho(L, e, m + 1, j)
```

Recherche dichotomique : version récursive

```
def dichot(L, e, i, j):  
    if i >= j: return False  
    m = (i + j) // 2 # milieu de i et j  
    if L[m] == e: return True  
    if e < L[m]: # il faut chercher à gauche de m  
        return dichot(L, e, i, m)  
    else: # il faut chercher à droite de m  
        return dichot(L, e, m + 1, j)
```

Utilisation :

```
In [4]: L = [1, 3, 4, 7, 9]  
  
In [5]: dichot(L, 7, 0, len(L) - 1)  
Out[5]: True  
  
In [6]: dichot(L, 8, 0, len(L) - 1)  
Out[6]: False
```


Question

Quel est le nombre d'appels récurifs en fonction de $n = \text{len}(L)$?

Question

Quel est le nombre d'appels récur­sifs en fonction de $n = \text{len}(L)$?

A chaque exécution, on divise au moins par deux la zone de recherche.

Au bout de deux exécutions, elle sera divisée par 4, puis 8, ... et 2^p au bout de p exécutions.

Question

Quel est le nombre d'appels récur­sifs en fonction de $n = \text{len}(L)$?

A chaque exécution, on divise au moins par deux la zone de recherche.

Au bout de deux exécutions, elle sera divisée par 4, puis 8, ... et 2^p au bout de p exécutions.

Donc, au bout de p exécutions, le nombre d'éléments de la zone de recherche est au plus :

$$\frac{n}{2^p}$$

Recherche dichotomique

Au bout de p appels récurifs, le nombre d'éléments de la zone de recherche est au plus $\frac{n}{2^p}$.

Quand $\frac{n}{2^p} \leq 1$, i.e $p \geq \log_2(n)$, la fonction s'arrête.

Recherche dichotomique

Au bout de p appels récurifs, le nombre d'éléments de la zone de recherche est au plus $\frac{n}{2^p}$.

Quand $\frac{n}{2^p} \leq 1$, i.e $p \geq \log_2(n)$, la fonction s'arrête.

Donc il y a au plus $\lceil \log_2(n) \rceil$ appels récurifs.

Comme chacun de ces appels effectue un nombre constant d'opérations, la complexité de la méthode par dichotomie est :

$$O(\log(n))$$

Trier une liste

On a donc besoin de savoir comment trier une liste, pour pouvoir utiliser dichotomie.

Question

Comment trier une liste ?

(Comment faites-vous pour trier votre main dans un jeu de cartes ?)

Tri par insertion

Le **tri par insertion** parcourt les indices i de L en conservant $L[:i]$ triée et en insérant $L[i]$ au bon endroit dans $L[:i]$

[5, 1, -4, 2, -8, 7]

Tri par insertion

Le **tri par insertion** parcourt les indices i de L en conservant $L[:i]$ triée et en insérant $L[i]$ au bon endroit dans $L[:i]$

[5, 1, -4, 2, -8, 7]

Tri par insertion

Le **tri par insertion** parcourt les indices i de L en conservant $L[:i]$ triée et en insérant $L[i]$ au bon endroit dans $L[:i]$

[1, 5, -4, 2, -8, 7]

Tri par insertion

Le **tri par insertion** parcourt les indices i de L en conservant $L[:i]$ triée et en insérant $L[i]$ au bon endroit dans $L[:i]$

`[-4, 1, 5, 2, -8, 7]`

Tri par insertion

Le **tri par insertion** parcourt les indices i de L en conservant $L[:i]$ triée et en insérant $L[i]$ au bon endroit dans $L[:i]$

`[-4, 1, 2, 5, -8, 7]`

Tri par insertion

Le **tri par insertion** parcourt les indices i de L en conservant $L[:i]$ triée et en insérant $L[i]$ au bon endroit dans $L[:i]$

`[-8, -4, 1, 2, 5, 7]`

Tri par insertion

Le **tri par insertion** parcourt les indices i de L en conservant $L[:i]$ triée et en insérant $L[i]$ au bon endroit dans $L[:i]$

`[-8, -4, 1, 2, 5, 7]`

Tri par insertion

On va se servir d'une fonction récursive `insere(L, i)` qui :

- 1 Suppose $L[:i]$ triée.
- 2 Met $L[i]$ à sa bonne place pour que $L[:i+1]$ devienne triée.

Tri par insertion

On va se servir d'une fonction récursive `insere(L, i)` qui :

- 1 Suppose `L[:i]` triée.
- 2 Met `L[i]` à sa bonne place pour que `L[:i+1]` devienne triée.

Fonctionnement de `insere(L, i)` :

- 1 Si `L[i-1] <= L[i]` :

Tri par insertion

On va se servir d'une fonction récursive `insere(L, i)` qui :

- 1 Suppose $L[:i]$ triée.
- 2 Met $L[i]$ à sa bonne place pour que $L[:i+1]$ devienne triée.

Fonctionnement de `insere(L, i)` :

- 1 Si $L[i-1] \leq L[i]$: $L[i]$ est à sa bonne position, on ne fait rien.
- 2 Si $L[i-1] > L[i]$:

Tri par insertion

On va se servir d'une fonction récursive `insere(L, i)` qui :

- 1 Suppose $L[:i]$ triée.
- 2 Met $L[i]$ à sa bonne place pour que $L[:i+1]$ devienne triée.

Fonctionnement de `insere(L, i)` :

- 1 Si $L[i-1] \leq L[i]$: $L[i]$ est à sa bonne position, on ne fait rien.
- 2 Si $L[i-1] > L[i]$: on peut échanger $L[i-1]$ et $L[i]$ puis appeler `insere(L, i-1)`.

Tri par insertion

Fonctionnement de `insere(L, i)` :

- 1 Si $i == 0$: il n'y a rien à faire (cas de base).
- 2 Si $L[i-1] \leq L[i]$: $L[i]$ est à sa bonne position, on ne fait rien.
- 3 Si $L[i-1] > L[i]$: on peut échanger $L[i-1]$ et $L[i]$ puis appeler `insere(L, i - 1)`.

```
def insere(L, i):  
    if i != 0 and L[i-1] > L[i]:  
        L[i], L[i-1] = L[i-1], L[i]  
        insere(L, i-1)
```

Tri par insertion

```
def insere(L, i):  
    if i != 0 and L[i-1] > L[i]:  
        L[i], L[i-1] = L[i-1], L[i]  
        insere(L, i-1)
```

```
def tri_insertion(L):  
    for i in range(len(L)):  
        insere(L, i)
```

Tri par insertion

```
def insere(L, i):  
    if i != 0 and L[i-1] > L[i]:  
        L[i], L[i-1] = L[i-1], L[i]  
        insere(L, i-1)
```

```
def tri_insertion(L):  
    for i in range(len(L)):  
        insere(L, i)
```

Remarque : pas de return, on modifie la liste en argument donc il n'y a pas besoin de renvoyer une nouvelle liste.

Tri par insertion

```
def insere(L, i):  
    if i != 0 and L[i-1] > L[i]:  
        L[i], L[i-1] = L[i-1], L[i]  
        insere(L, i-1)
```

```
def tri_insertion(L):  
    for i in range(len(L)):  
        insere(L, i)
```

Comment prouver que ce tri est correct ?

Tri par insertion

```
def insere(L, i):  
    if i != 0 and L[i-1] > L[i]:  
        L[i], L[i-1] = L[i-1], L[i]  
        insere(L, i-1)
```

```
def tri_insertion(L):  
    for i in range(len(L)):  
        insere(L, i)
```

Comment prouver que ce tri est correct ?

En montrant l'**invariant de boucle** :

H_i = « au début de la i ème itération de la boucle for,
L[0:i] est triée »

Tri par insertion

```
def insere(L, i):  
    if i != 0 and L[i-1] > L[i]:  
        L[i], L[i-1] = L[i-1], L[i]  
        insere(L, i-1)
```

```
def tri_insertion(L):  
    for i in range(len(L)):  
        insere(L, i)
```

Quelle est sa complexité dans le pire cas ?

Tri par insertion

```
def insere(L, i):  
    if i != 0 and L[i-1] > L[i]:  
        L[i], L[i-1] = L[i-1], L[i]  
        insere(L, i-1)
```

```
def tri_insertion(L):  
    for i in range(len(L)):  
        insere(L, i)
```

Quelle est sa complexité dans le pire cas ?

❶ `insere(L, i)` est en

Tri par insertion

```
def insere(L, i):  
    if i != 0 and L[i-1] > L[i]:  
        L[i], L[i-1] = L[i-1], L[i]  
        insere(L, i-1)
```

```
def tri_insertion(L):  
    for i in range(len(L)):  
        insere(L, i)
```

Quelle est sa complexité dans le pire cas ?

- 1 `insere(L, i)` est en $O(i)$
- 2 donc `tri_insertion(L)` est en

Tri par insertion

```
def insere(L, i):  
    if i != 0 and L[i-1] > L[i]:  
        L[i], L[i-1] = L[i-1], L[i]  
        insere(L, i-1)
```

```
def tri_insertion(L):  
    for i in range(len(L)):  
        insere(L, i)
```

Quelle est sa complexité dans le pire cas ?

- 1 $\text{insere}(L, i)$ est en $O(i)$
- 2 donc $\text{tri_insertion}(L)$ est en $O(1)+O(2)+\dots+O(n) = O(n^2)$.

On va voir des algorithmes plus efficaces.

Tri fusion

Le tri fusion sur une liste L consiste à :

- 1 Séparer L en deux listes $L1$ et $L2$ de même taille.

Le tri fusion sur une liste L consiste à :

- ① Séparer L en deux listes L1 et L2 de même taille.
- ② Trier récursivement L1 et L2 pour obtenir des listes triées L1' et L2'

Tri fusion

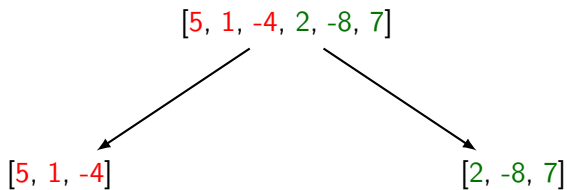
Le tri fusion sur une liste L consiste à :

- ① Séparer L en deux listes $L1$ et $L2$ de même taille.
- ② Trier récursivement $L1$ et $L2$ pour obtenir des listes triées $L1'$ et $L2'$
- ③ Fusionner $L1'$ et $L2'$ pour avoir un tri de L .

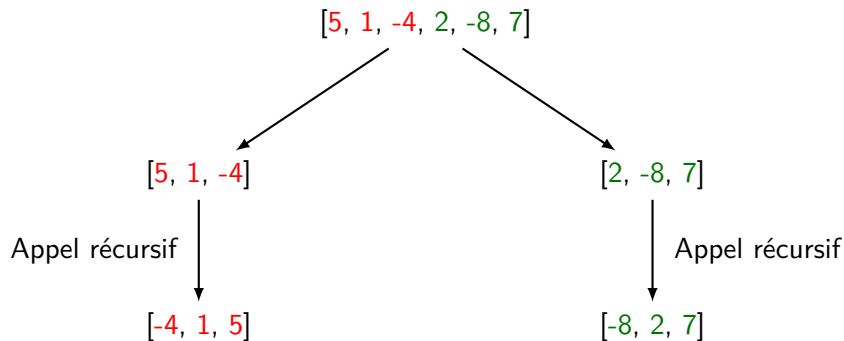
Tri fusion

[5, 1, -4, 2, -8, 7]

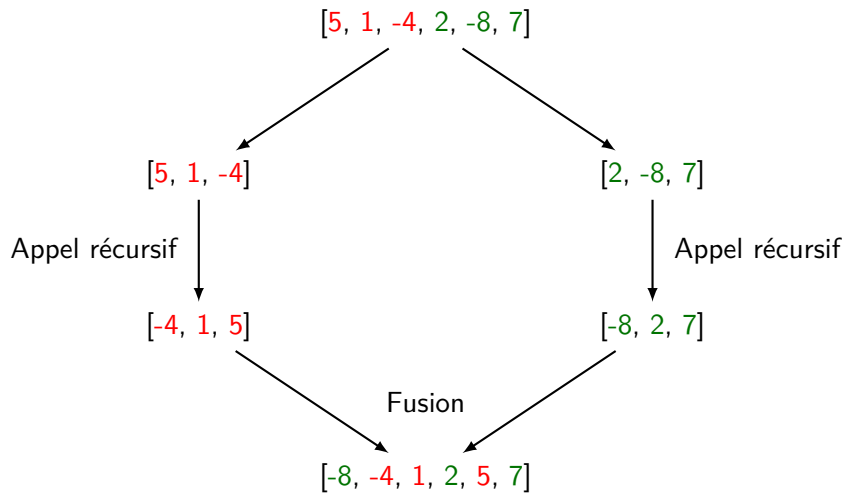
Tri fusion



Tri fusion



Tri fusion



On va se servir d'une fonction récursive `fusion(L1, L2)` qui :

- 1 Suppose L1 et L2 triées.
- 2 Renvoie une liste triée contenant les éléments des deux listes.

On va se servir d'une fonction récursive `fusion(L1, L2)` qui :

- 1 Suppose `L1` et `L2` triées.
- 2 Renvoie une liste triée contenant les éléments des deux listes.

Fonctionnement de `fusion(L1, L2)` :

- 1 Si `len(L1) == 0` :

On va se servir d'une fonction récursive `fusion(L1, L2)` qui :

- 1 Suppose L1 et L2 triées.
- 2 Renvoie une liste triée contenant les éléments des deux listes.

Fonctionnement de `fusion(L1, L2)` :

- 1 Si `len(L1) == 0` : renvoyer L2.
- 2 Si `len(L2) == 0` :

On va se servir d'une fonction récursive `fusion(L1, L2)` qui :

- 1 Suppose `L1` et `L2` triées.
- 2 Renvoie une liste triée contenant les éléments des deux listes.

Fonctionnement de `fusion(L1, L2)` :

- 1 Si `len(L1) == 0` : renvoyer `L2`.
- 2 Si `len(L2) == 0` : renvoyer `L1`.
- 3 Sinon :
 - a) Soit `m` le maximum de `L1` et `L2`.

On va se servir d'une fonction récursive `fusion(L1, L2)` qui :

- 1 Suppose `L1` et `L2` triées.
- 2 Renvoie une liste triée contenant les éléments des deux listes.

Fonctionnement de `fusion(L1, L2)` :

- 1 Si `len(L1) == 0` : renvoyer `L2`.
- 2 Si `len(L2) == 0` : renvoyer `L1`.
- 3 Sinon :
 - a) Soit `m` le maximum de `L1` et `L2`.
 - b) Supprimer `m`.

On va se servir d'une fonction récursive `fusion(L1, L2)` qui :

- 1 Suppose L1 et L2 triées.
- 2 Renvoie une liste triée contenant les éléments des deux listes.

Fonctionnement de `fusion(L1, L2)` :

- 1 Si `len(L1) == 0` : renvoyer L2.
- 2 Si `len(L2) == 0` : renvoyer L1.
- 3 Sinon :
 - a) Soit `m` le maximum de L1 et L2.
 - b) Supprimer `m`.
 - c) Fusionner récursivement L1 et L2 (où `m` a été supprimé).

On va se servir d'une fonction récursive `fusion(L1, L2)` qui :

- 1 Suppose `L1` et `L2` triées.
- 2 Renvoie une liste triée contenant les éléments des deux listes.

Fonctionnement de `fusion(L1, L2)` :

- 1 Si `len(L1) == 0` : renvoyer `L2`.
- 2 Si `len(L2) == 0` : renvoyer `L1`.
- 3 Sinon :
 - a) Soit `m` le maximum de `L1` et `L2`.
 - b) Supprimer `m`.
 - c) Fusionner récursivement `L1` et `L2` (où `m` a été supprimé).
 - d) Rajouter `m` au résultat de la fusion.

Tri fusion

```
def fusion(L1, L2):  
    if len(L1) == 0: return L2  
    if len(L2) == 0: return L1  
    if L1[-1] > L2[-1]: m = L1.pop()  
    else: m = L2.pop()  
    L = fusion(L1, L2)  
    L.append(m)  
    return L
```

Tri fusion

```
def fusion(L1, L2):  
    if len(L1) == 0: return L2  
    if len(L2) == 0: return L1  
    if L1[-1] > L2[-1]: m = L1.pop()  
    else: m = L2.pop()  
    L = fusion(L1, L2)  
    L.append(m)  
    return L
```

Complexité :

Tri fusion

```
def fusion(L1, L2):  
    if len(L1) == 0: return L2  
    if len(L2) == 0: return L1  
    if L1[-1] > L2[-1]: m = L1.pop()  
    else: m = L2.pop()  
    L = fusion(L1, L2)  
    L.append(m)  
    return L
```

Complexité :

Soit $n = \text{len}(L1) + \text{len}(L2)$.

`fusion(L1, L2)` effectue $O(n)$ appels récurifs (car à chaque appel on enlève un élément) et chaque appel récursif effectue un nombre constant d'opérations.

Donc la complexité de `fusion(L1, L2)` est $O(n)$.

Tri fusion

Fonctionnement de `tri_fusion(L)` :

- 1 Si `len(L) <= 1` :

Tri fusion

Fonctionnement de `tri_fusion(L)` :

- 1 Si `len(L) <= 1` : L est déjà triée.
- 2 Séparer L en deux listes de même taille L1 et L2.
- 3 Trier récursivement L1 et L2.
- 4 Les fusionner.

Tri fusion

Fonctionnement de `tri_fusion(L)` :

- 1 Si `len(L) <= 1` : L est déjà triée.
- 2 Séparer L en deux listes de même taille L1 et L2.
- 3 Trier récursivement L1 et L2.
- 4 Les fusionner.

```
def tri_fusion(L):  
    if len(L) <= 1: return L  
    L1, L2 = L[: len(L)//2], L[len(L)//2 :]  
    return fusion(tri_fusion(L1), tri_fusion(L2))
```

Tri fusion

```
def tri_fusion(L):  
    if len(L) <= 1: return L  
    L1, L2 = L[: len(L)//2], L[len(L)//2 :]  
    return fusion(tri_fusion(L1), tri_fusion(L2))
```

Question

Comment prouver que tri_fusion(L) trie L ?

Tri fusion

```
def tri_fusion(L):  
    if len(L) <= 1: return L  
    L1, L2 = L[: len(L)//2], L[len(L)//2 :]  
    return fusion(tri_fusion(L1), tri_fusion(L2))
```

Question

Comment prouver que `tri_fusion(L)` trie `L` ?

Par récurrence sur la taille de `L` : par hypothèse de récurrence, les appels `tri_fusion(L1)` et `tri_fusion(L2)` trient `L1` et `L2` et on en déduit que `tri_fusion(L)` trie `L`.

Question

Quelle est la complexité dans le pire cas de `tri_fusion(L)` ?

Notons $C(n)$ cette complexité pour $n = \text{len}(L)$.

Question

Quelle est la complexité dans le pire cas de `tri_fusion(L)` ?

Notons $C(n)$ cette complexité pour $n = \text{len}(L)$.

`tri_fusion(L)` effectue :

- 1 appels récursifs sur $L1$ et $L2$: complexité $2 \times C(\frac{n}{2})$.
- 2 fusion des deux listes : complexité n .

D'où : $C(n) = n + 2C(\frac{n}{2})$.

$$C(n) = n + 2C\left(\frac{n}{2}\right)$$

En appliquant cette inégalité sur $C\left(\frac{n}{2}\right)$:

$$C(n) = n + 2\left(\frac{n}{2} + 2C\left(\frac{n}{4}\right)\right) = n + n + 4C\left(\frac{n}{4}\right)$$

...

$$C(n) = n + 2C\left(\frac{n}{2}\right)$$

En appliquant cette inégalité sur $C\left(\frac{n}{2}\right)$:

$$C(n) = n + 2\left(\frac{n}{2} + 2C\left(\frac{n}{4}\right)\right) = n + n + 4C\left(\frac{n}{4}\right)$$

...

$$C(n) = \underbrace{n + n + \dots + n}_k + 2^k C\left(\frac{n}{2^k}\right)$$

Tri fusion

$$C(n) = n + 2C\left(\frac{n}{2}\right)$$

En appliquant cette inégalité sur $C\left(\frac{n}{2}\right)$:

$$C(n) = n + 2\left(\frac{n}{2} + 2C\left(\frac{n}{4}\right)\right) = n + n + 4C\left(\frac{n}{4}\right)$$

...

$$C(n) = \underbrace{n + n + \dots + n}_k + 2^k C\left(\frac{n}{2^k}\right)$$

Pour $k = \log_2(n)$:

$$C(n) = n \log_2(n) + 2^{\log_2(n)} C\left(\frac{n}{2^{\log_2(n)}}\right)$$

$$C(n) = n \log_2(n) + nC(1) = O(n \log(n))$$

Donc la complexité du tri fusion est $\boxed{O(n \log(n))}$.

Tri rapide (Quicksort)

Le tri rapide, sur une liste L , consiste à :

- 1 Séparer L en deux listes L_1 et L_2 : les éléments plus petits que $L[0]$ et les éléments plus grand que $L[0]$.

Tri rapide (Quicksort)

Le tri rapide, sur une liste L , consiste à :

- 1 Séparer L en deux listes $L1$ et $L2$: les éléments plus petits que $L[0]$ et les éléments plus grand que $L[0]$.
- 2 Trier récursivement $L1$ et $L2$.

Tri rapide (Quicksort)

Le tri rapide, sur une liste L , consiste à :

- 1 Séparer L en deux listes $L1$ et $L2$: les éléments plus petits que $L[0]$ et les éléments plus grand que $L[0]$.
- 2 Trier récursivement $L1$ et $L2$.
- 3 Renvoyer la liste triée $L1 + [L[0]] + L2$.

Tri rapide (Quicksort)

Le tri rapide, sur une liste L , consiste à :

- 1 Séparer L en deux listes $L1$ et $L2$: les éléments plus petits que $L[0]$ et les éléments plus grand que $L[0]$.
- 2 Trier récursivement $L1$ et $L2$.
- 3 Renvoyer la liste triée $L1 + [L[0]] + L2$.

```
def tri_rapide(L):  
    if len(L) <= 1: return L  
    L1, L2 = [], []  
    for i in range(1, len(L)):  
        if L[i] < L[0]: L1.append(L[i])  
        else: L2.append(L[i])  
    return tri_rapide(L1) + [L[0]] + tri_rapide(L2)
```

Tri rapide (Quicksort)

Remarque : on peut aussi l'écrire en deux lignes avec des « listes par compréhension »...

```
def tri_rapide(L):  
    if len(L) <= 1: return L  
    return tri_rapide([e for e in L if e < L[0]])  
+ [L[0]] + tri_rapide([e for e in L if e > L[0]])
```

Tri rapide

Question

Quelle est la complexité de `tri_rapide` ?

Question

Quelle est la complexité de `tri_rapide` ?

- Dans le meilleur des cas : L1 et L2 sont de taille $\frac{n}{2}$. La complexité est $O(n \log(n))$, comme pour le tri fusion.
- Dans le pire cas : la taille de L1 (ou L2) est systématiquement égale à $n - 1$. On a alors une complexité (à une constante près) :

$$C(n) = C(n - 1) + n = 1 + 2 + \dots + n = \Theta(n^2)$$

Récapitulatif

Comparaison des algorithmes de tri sur une liste de taille n :

	Meilleur cas	Pire cas
Tri par insertion	$O(n)$	$O(n^2)$
Tri fusion	$O(n \log(n))$	$O(n \log(n))$
Tri rapide	$O(n \log(n))$	$O(n^2)$

Question

Comment trouver la **médiane** d'une liste L de nombres, c'est à dire l'élément m de L tel qu'il y ait autant d'éléments supérieurs à m que d'éléments inférieurs ?

Exemple : la médiane de $[1, 0, 6, 27, 8, -10, 21]$ est 6. En effet il y a trois éléments inférieurs à 6 et trois éléments supérieurs à 6.

Application des tris

- 1ère possibilité : tester, pour chaque élément m de L , si m est la médiane.

Application des tris

- ❶ 1ère possibilité : tester, pour chaque élément m de L , si m est la médiane.
En $O(n^2)$.
- ❷ 2ème possibilité : trier L puis sélectionner l'élément « du milieu ».

Application des tris

- ❶ 1ère possibilité : tester, pour chaque élément m de L , si m est la médiane.
En $O(n^2)$.
- ❷ 2ème possibilité : trier L puis sélectionner l'élément « du milieu ».
En $O(n \log(n))$

Application des tris

- ❶ 1ère possibilité : tester, pour chaque élément m de L , si m est la médiane.
En $O(n^2)$.
- ❷ 2ème possibilité : trier L puis sélectionner l'élément « du milieu ».
En $O(n \log(n))$

```
def mediane(L):  
    L1 = tri_fusion(L)  
    return L1[len(L1)//2]
```

```
In [54]: mediane([3, 2, 6, 1, 8, 4, 5])  
Out[54]: 4
```

Égalité à permutation près

Exercice

Écrire une fonction `egal(L1, L2)` déterminant si deux listes d'entiers `L1` et `L2` sont égales à permutation près.

Exemples : `egal([1, 2, 3], [2, 1, 3])` doit renvoyer `True`.
`egal([1, 2, 4], [2, 1, 3])` doit renvoyer `False`.

Tris de chaînes de caractères

Nous avons vu des tris sur des listes de nombres.

Ils permettent aussi de trier des chaînes de caractères selon l'ordre du dictionnaire (alphabétique).

Tris de chaînes de caractères

Nous avons vu des tris sur des listes de nombres.

Ils permettent aussi de trier des chaînes de caractères selon l'ordre du dictionnaire (alphabétique).

```
In [58]: L = ["MPSI", "PCSI", "PSI", "PC", "MP"]
```

```
In [59]: tri_insertion(L)
```

```
In [60]: L
```

```
Out[60]: ['MP', 'MPSI', 'PC', 'PCSI', 'PSI']
```

```
In [61]: "PSI" > "PC"
```

```
Out[61]: True
```