Informatique Pour Tous

Cours 22 – Tri de données numériques

Qu'il s'agisse d'algorithmes d'aide décisionnelle (trouver les conditions maximisant un critère), de *machine learning* (déterminer la ou les meilleures solutions au regard d'un critère), ou encore de classement des éléments d'une liste (par exemple classer les articles d'un site marchand en fonction d'un critère, comme le prix ou la note), les algorithmes de tri sont extrêmement courants dans le quotidien informatique.

Rappelons également qu'une fois une liste ou une table triée par ordre croissant d'un critère, il devient possible de rechercher des éléments par dichotomie, de façon nettement plus efficace qu'une recherche dans une liste non triée. Aussi, lorsqu'une liste de grande taille est peu souvent éditée mais souvent lue pour y rechercher des éléments, le tri par ordre croissant permet un gain de temps processeur notable.

Cette année, nous nous intéresserons uniquement au tri de données numériques, contenues dans une seule liste de nombres (ou dans un vecteur *Numpy array* de dimension 1). Nous restreindrons ainsi nos méthodes de tri aux seuls tris **comparatifs**, c'est-à-dire des tris où l'on compare les éléments entre eux, ici à l'aide d'inégalités.

A / Quelques rappels

A.1 / Complexité d'un algorithme de tri

Les algorithmes de tri s'appliquent souvent à des listes de grande taille (*big datas*), et l'optimisation de leur complexité est alors primordiale. Cependant, le nombre d'opérations lors d'un tri ne dépend pas seulement de la taille de la liste. Reprenons l'exemple d'un paquet de cartes à trier par ordre croissant : suivant que le jeu est presque totalement trié (seules certaines cartes sont dérangées), partiellement trié (déjà organisé par couleurs après un jeu de cartes) ou totalement désorganisé, le nombre d'opérations à réaliser (et donc le temps de tri) ne sera pas le même.

Ainsi, suivant le type d'algorithme retenu, on distinguera trois types de complexités :

- La complexité dans le meilleur des cas (on regardera alors combien d'itérations notre méthode de tri réalise sur une liste déjà triée)
- La complexité dans le pire des cas (pour la plupart des algorithmes, c'est lorsque la liste initiale est rangée par ordre décroissant)
- La complexité moyenne (pour des données initiales réparties aléatoirement, c'est elle qu'on caractérise « empiriquement »)

A.2 / Tri en place ou non

Soit une liste de nombres `L = [3,7,5,2,1]` déclarée comme variable globale. Deux fonctions (en réalité une fonction et une méthode) de tri sont disponibles nativement sur *Python* :

- `sorted(L)` renvoie (retourne) `[1,2,3,5,7]`

  Après cet appel, `L` vaut `[3,7,5,2,1]`

  Ce n'est pas un tri en place. Il s'agit d'une FONCTION.

- `L.sort()` renvoie rien `(None)`. Il s'agit d'une METHODE.

  Après cet appel, `L` vaut `[1,2,3,5,7]`

  Il s'agit d'un tri **en place**

Supposons une fonction `Tri(L)` prenant en argument une liste de nombre `L` et réalisant un algorithme de tri. Dans la séquence précédente sur les piles, nous avons utilisé des fonctions **procédurales** (ou procédures), qui affectaient la liste en argument sans effectuer de `return`.

- Le tri est **en place** si `Tri(L)` affecte `L` en global, sans la renvoyer

  Remarque : la **seule et unique** opération autorisée pour trier la liste, pour un tri **en place**, est la permutation d'éléments entre eux : `L[i],L[j] = L[j],L[i]`

- Dans le cas contraire, `Tri(L)` crée une autre variable (ex : `L2 = []`) qu'il édite et renvoie en fin d'algo.

  Remarque : Pour éviter de modifier `L` à l'extérieur de la fonction, je saurai que trop vous conseiller de **commencer votre fonction** par faire une copie ! `L_c = L.copy()  # ou L_c = L[:]`

Remarque : l'utilisation d'un algorithme **en place** présente l'intérêt de peu utiliser la mémoire vive. En effet, rappelons que les valeurs des variables *Python* sont stockées dans la RAM de l'ordinateur, et donc l'utilisation de variables supplémentaires (en plus de la liste à trier) induit forcément une complexité spatiale (quantité d'octets de RAM utilisés par l'algorithme) accrue.

B / Algorithmes intuitifs mais peu « performants »

B.1 / Tri par *SELECTION*

Il s'agit du plus intuitif des algorithmes de tri. Exemple de structure :

```
L_a_trier = L.copy()                    # paquet initial (non trié), à l'endroit (cartes visibles)
L_triee = []                            # paquet vide (va se remplir), retourné à l'envers (dos des cartes)
while len(L_a_trier) > 0 :              # on balaie le paquet à trier tant qu'il n'est pas vidé
    index_element_mini = 0              # l'indice du 1er élément de L_a_trier
    for i in range(1,len(L_a_trier)) :  # à la recherche de
        if L_a_trier[i] < L[index_element_mini] :   # (la position de)
            index_element_mini = i      # la plus petite carte
    element = L_a_trier.pop(index_element_mini)     # on déplace l'élément correspondant
                                        # au sommet de l'autre tas, face
    L_triee.append(element)             # cachée (pour les avoir dans
                                        # l'ordre à la fin)
```

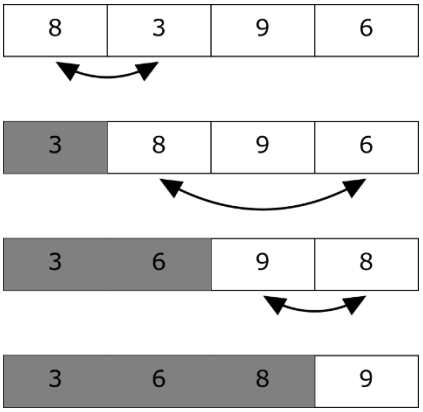
Remarque : ainsi décrit, l'algorithme est stable, mais n'est pas en place.

- L'astuce, pour écrire un tri simple **en place**, consiste à séparer artificiellement la liste en 2, à l'aide d'un index `i` : les éléments **avant `i`** sont déjà triés, les éléments **après `i`** ne le sont pas encore.
  → On va alors balayer `i` du début à la fin de la liste, quand il est arrivé à la fin, la liste est totalement triée.

On peut toutefois l'écrire en place, par exemple en supposant que pour toute $i^{ème}$ carte du paquet, les précédentes sont triées mais pas les suivantes :

```
Pour chaque élément L[i] dans la liste d'entrée (0 ≤ i ≤ N − 2)     # on balaie jusqu'à l'avant-dernière carte
    Soit index_element_mini = i
    Pour chaque élément L[j] de la liste à trier (i < j ≤ N − 1)    # à partir de cette carte, on balaie
                                                                    # la suite du paquet (jusqu'à la fin)
        Si L[j] < L[index_element_mini]                             # à la recherche de (la position de)
            index_element_mini = j                                  # la plus petite carte
    On permute L[index_element_mini] avec L[i]                      # on permute les 2 cartes
```

Exemple : liste d'entrée `L = [8,3,9,6]`



Crédit : cours IPT lycée Lafayette

Valeur clé : 8 (position 0) :
- 3 < 8 ? oui donc `index_element_mini` ← 1
- 9 < 3 ?
- 6 < 3 ?
- → On permute les positions 0 (clé) et 1 (index mini)

Valeur clé : 8 (position 1) :
- 9 < 8 ?
- 6 < 8 ? oui donc `index_element_mini` ← 3
- → On permute les positions 1 (clé) et 3 (index mini)

Valeur clé : 9 (position 2) :
- 8 < 9 ? oui donc `index_element_mini` ← 3
- → On permute les positions 2 (clé) et 3 (index mini)

Cela donne le code suivant :

```
for i in range(len(L)-1) :
    index_element_mini = i
    for j in range(i+1,len(L)) :
        if  L[j] < L[index_element_mini] :
            index_element_mini = j
    L[index_element_mini] , L[i] = L[i] , L[index_element_mini]
```

Soit N le nombre d'éléments dans la liste à trier, et C_N le nombre d'opérations à réaliser pour trier la liste. On remarque qu'en ajoutant un élément à la liste, le nombre d'opérations varie suivant une récurrence :

$$C_{N+1} = C_N + N + 1$$

Ainsi, $C_N = \sum_{i=1}^{N} i = \frac{N(N+1)}{2} = O(N^2)$, on a donc une complexité quadratique, que ce soit dans le meilleur ou le pire des cas (la complexité ne dépend pas du tri initial de la liste). Ceci est cohérent car le code s'appuie sur 2 boucles imbriquées.

Remarque : cet algorithme effectue un grand nombre de comparaisons entre éléments, en revanche il effectue peu de permutations ($O(N)$ permutations). Dans certains cas où les données sont peu coûteuses à comparer mais très coûteuses à déplacer, il peut présenter un intérêt.

B.2 / Tri par *INSERTION*

Le tri par **insertion** est le tri que nous appliquons « naturellement » pour trier un jeu de cartes. Pour bien comprendre, faisons cette fois (on n'est pas obligés !) appel à une **fonction auxiliaire**.

Cette fonction `insertion(a, L)` prend en argument une liste **déjà triée** (ordre croissant) `L` et cherche l'index auquel insérer un élément `a` pour que l'ordre croissant soit maintenu.

```
def insertion(a, L) :
    for i in range(len(L)) :
        if a < L[i] :
            return i
    return len(L)          # cette dernière ligne n'est lue que si tous les éléments de la liste sont < a
                           # auquel cas il faudra insérer a en dernier élément (index = len(L))
```

Avec l'aide de cette fonction, le tri par insertion peut simplement s'écrire :

```
L_a_trier = L.copy()                    # paquet initial (non trié), à l'endroit (cartes visibles)
L_triee = []                            # paquet vide (va se remplir), retourné à l'envers (dos des cartes)
while len(L_a_trier) > 0 :              # on balaie le paquet à trier tant qu'il n'est pas vidé
    element = L_a_trier.pop(0)          # on dépile L_a_trier par la gauche
    index_insert = insertion(element, L_triee)  # on détermine où l'insérer dans la liste triée
    L_triee.insert(index_insert, element)       # attention à l'ordre des éléments
```

Remarque : ce tri n'est pas **en place**. Pour ce faire, on modifie légèrement l'algorithme. Comme tout à l'heure, nous séparons artificiellement le paquet en `L[:i]` les cartes **déjà triées**, et `L[i+1:]` les cartes qu'il **reste à trier**.

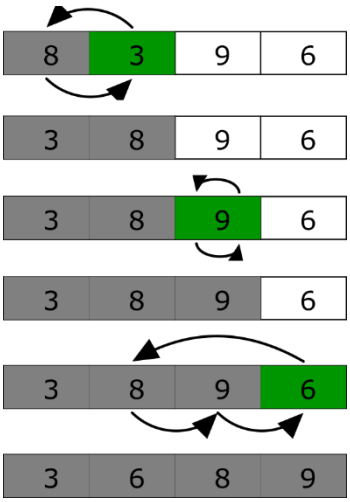
Pour chaque élément `L[i]` dans la liste d'entrée ($1 \leq i \leq N-1$) # on balaie progressivement le paquet

On regarde tous les éléments précédents `L[j]` en remontant à l'envers *j* ↘ de *i* à 0

Chaque fois qu'on a `L[j] < L[j-1]` # la carte suivante < la précédente (ordre inversé)

On permute `L[j]` et `L[j-1]` # on permute les 2

Exemple avec une liste d'entrée `L = [8,3,9,6]`



Valeur clé : 3 (position 1) :
- 3 < 8 ? oui donc on permute les positions 0 et 1

Valeur clé : 9 (position 2) :
- 9 < 8 ?
- 9 < 3 ?

Valeur clé : 6 (position 3) :
- 6 < 9 ? oui donc on permute les positions 2 (9) et 3 (6)
- 6 < 8 ? oui donc on permute les positions 1 (8) et 2 (6)

Remarque : ces 2 étapes sont équivalentes à insérer 6 devant 8.

Crédit : cours IPT lycée Lafayette

Cela donne le code suivant :

```
for i in range(1,len(L)) :
    for j in range(i,0,-1) :
        if  L[j] < L[j-1]:
            L[j] , L[j-1] = L[j-1] , L[j]
```

Soit N le nombre d'éléments dans la liste à trier, et C_N le nombre d'opérations à réaliser pour trier la liste. On remarque qu'en ajoutant un élément à la liste, le nombre d'opérations varie dans le **pire des cas** suivant une récurrence :

$$C_{N+1} = C_N + N - 1$$

Ainsi, $C_N = \sum_{i=1}^{N-1} i = \frac{N(N-1)}{2} = O(N^2)$ de complexité quadratique, dans le pire des cas (liste initiale à l'envers). On aurait pu s'en douter car il y a 2 boucles imbriquées : oui, attention, quand il y a un appel de fonction il faut compter la complexité de la fonction ! Ici on a *N* appels de la fonction `insertion`, qui est elle-même de complexité *O(N)*.

Remarque : la complexité dans le meilleur des cas est en $O(N)$, si la liste initiale est totalement triée. En revanche, la complexité moyenne reste quadratique, comme la complexité dans le pire des cas. Cet algorithme est donc performant dans des cas où la liste initiale est déjà presque triée. Autrement dit, plus le jeu de cartes est mélangé, moins cette méthode de tri est efficace.

B.2 / Tri à *BULLES*

Nous avons déjà rencontré cet algorithme de tri dans le cours cours/TD 18. Nous en rappelons le principe ici pour compléter la liste des tris « intuitifs ».

Le tri à bulles est un algorithme de tri « quadratique », peu optimisé, qui se fait **en place**. Nous allons balayer plusieurs fois la liste et comparer chaque élément à l'élément qui le suit directement, et faire si besoin « remonter » (vers la gauche de la liste, pour un tri par ordre croissant) l'élément le plus petit par permutation des deux éléments. Le nom évoquerait des bulles d'air qui remontent à la surface de façon successive.

L'algorithme s'appuie sur deux boucles imbriquées. Comprenons-le au travers d'un exemple : en veut trier par ordre croissant la liste de nombres suivante : `L = [2, 5, 4, 3, 1].`
Au pire des cas (c'est, volontairement le cas dans l'exemple ici), le plus petit élément est à la fin, il va donc falloir faire 4 passages successifs (c'est-à-dire `len(L)-1`).

À chacun de ces passages `i`, nous allons balayer les éléments, du 1er jusqu'au i^{ème} en partant de la fin (ce que je vous écris ici, c'est en notations mathématiques, il faudra réfléchir aux index en *Python*).

Lors de ce balayage, on compare chaque élément avec le suivant :

- Si ces éléments sont déjà dans « le bon ordre » (localement), on les laisse tels quel
- Sinon, on permute ces deux éléments

Exemples :

- Premier passage :
  [2, 5, 4, 3, 1], on compare 2 et 5 et on ne fait rien car 5 > 2 ;
  [2, 5, 4, 3, 1], on compare 5 et 4 et on les permute car 5 < 4 ;
  [2, 4, 5, 3, 1], on compare 5 et 3 et on les permute ;
  [2, 4, 3, 5, 1], on compare 5 et 1 et on les permute ;
  ➔ Ainsi on a `L` = [2, 4, 3, 1, 5], fin du premier passage.

- Deuxième passage :
  [2, 4, 3, 1, 5], on compare 2 et 4 et on ne fait rien ;
  [2, 4, 3, 1, 5], on compare 4 et 3 et on les permute ;
  [2, 3, 4, 1, 5], on compare 4 et 1 et on les permute ;
  ➔ Ainsi, on a `L` = [2, 3, 1, 4, 5], fin du deuxième passage.

- Troisième passage :
  [2, 3, 1, 4, 5], on compare 2 et 3 et on ne fait rien ;
  [2, 3, 1, 4, 5], on compare 3 et 1 et on les permute ;
  ➔ [2, 1, 3, 4, 5], fin du troisième passage.

- Quatrième passage :
  [2, 1, 3, 4, 5], on compare 2 et 1 et on les permute ;
  ➔ [1, 2, 3, 4, 5], fin du quatrième passage.

➔ Fin de l'algorithme

Cela donne le code suivant :

```
def tri_bulles(L):
    N = len(L)
    for i in range(1,N):
        for j in range(N-i):  # bien réfléchir aux bornes,
                              #attention au L[j+1] qui ne doit pas renvoyer d'index out of bounds
            if L[j] > L[j+1]:
                L[j], L[j+1] = L[j+1], L[j]  # permutation de 2 éléments successifs
    # la fonction n'a pas de return (procédurale, tri en place)
```

C / Algorithmes s'appuyant sur une méthode « diviser pour régner », méthode récursive

Vous avez découvert le principe des méthodes ***« diviser pour régner »*** avec la dichotomie. Là où une méthode « naïve » cherchait un élément à tâtons, en testant chacun des éléments d'une liste (complexité $O(N)$), la dichotomie découpe à chaque itération des segments et divise le problème en deux, d'où sa complexité logarithmique ($O(\log(N))$). Les algorithmes de tri peuvent suivre la même logique.

Arrêtons-nous une seconde pour une preuve de principe : nous venons de voir que les algorithmes de tri « naïfs » sont de complexité $O(N^2)$. Ainsi, si nous divisons la liste en deux listes de taille $N/2$, trier chacune des deux demi-listes séparément devrait prendre $N^2/4 + N^2/4 = N^2/2$ opérations. Si l'on découpe à nouveau chacune de ces deux demi-listes en deux quarts de listes, on devrait avoir au total $(N^2/16) \times 4 = N^2/4$ opérations, et ainsi de suite jusqu'à la limite ou on aura découpé la liste initiale de taille N en N listes de taille 1. Une fois cette opération réalisée, nous allons devoir rassembler les morceaux pour les ré-organiser en une grande liste triée de taille *N*. Comme pour la dichotomie, qui faisait passer la complexité de $O(N)$ à $O(\log(N))$, les algorithmes de tri s'appuyant sur une méthode « diviser pour régner » vont faire passer la complexité de $O(N^2)$ à $O(N\log(N))$.

Remarque : qu'ils soient en place ou non (fonction procédurale ou non), les algorithmes suivants se décrivent par ***récursivité***, c'est-à-dire qu'ils s'appellent eux-mêmes pour le fameux découpage de la liste initiale.

C.1 / Tri *FUSION*

Afin de décrire cet algorithme, nous allons devoir poser deux fonctions préalables. Pour le tri **fusion**, nous abandonnons tout projet de le décrire **en place**, nous n'utiliserons donc pas de fonction procédurale.

➢ C.1.i / la fonction découpe (`L`)

Cette fonction découpe littéralement la liste `L` en deux. Il existe plusieurs façons de faire, par exemple :

- Couper au niveau du dernier élément de `L`

  `decoupe( [1,2,4] )` renverrait : `[1,2] , [4]` (2 listes)

  `decoupe( [1] )` renverrait : `[1] , []` ou bien `[] , [1]` (selon la façon dont elle est codée)

- Couper au niveau du 1er élément de `L`

  `decoupe( [1,2,4] )` renverrait : `[1] , [2, 4]`

  `decoupe( [1] )` renverrait : `[] , [1]` ou bien `[1] , []`

- Couper au niveau du milieu de `L` pour avoir 2 listes de tailles sensiblement identiques

  `decoupe( [1,2,4, -1] )` renverrait : `[1,2] , [4, -1]`

  `decoupe( [1,4,0] )` renverrait : `[1] , [4,0]` ou bien `[1,4] , [0]`

Remarque : pour la suite de ce cours, nous supposerons que la découpe se fait au milieu.

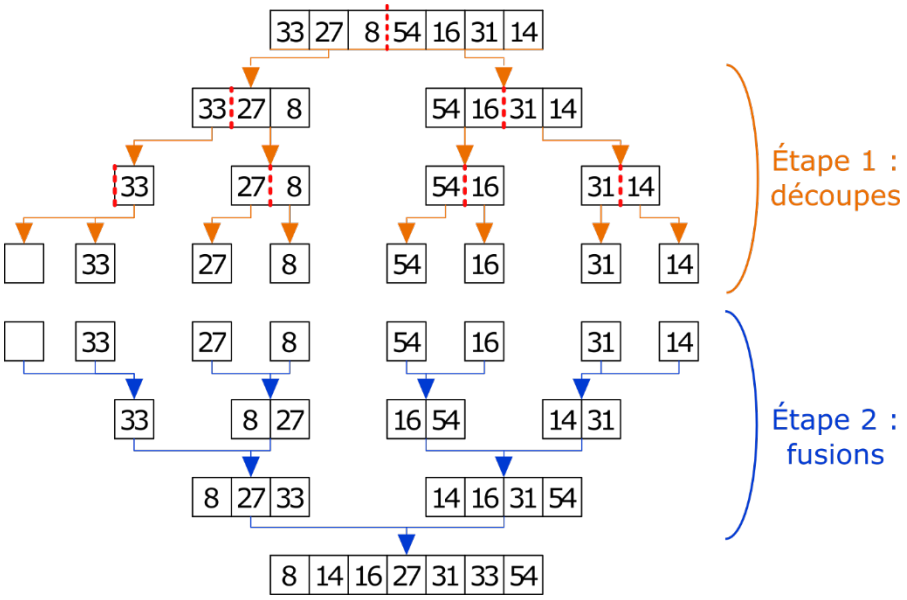
➢ C.2.ii / la fonction fusion (`L1 , L2`)

Cette fonction reçoit deux listes `L1` et `L2` triées, et renvoie une liste + longue (de taille `len(L1)+len(L2)`) issue de la fusion des deux listes d'entrée en une seule liste triée. Exemples :

```
fusion( [1,3,7] , [4, 6] ) renverrait  [1, 3, 4, 6, 7]
fusion( [0, 4] , [-1, 0] ) renverrait  [-1, 0, 0, 4]  ou [-1, 0, 0, 4]
```

➢ C.2.iii / le principe de l'algorithme de tri fusion

Le principe est de découper, dans un premier temps, la liste initiale en listes unitaires (d'où l'intérêt, visible sur le schéma ci-dessous, d'avoir choisi la découpe au milieu, qui permet de découper « en parallèle » là où la découpe d'un côté ou de l'autre se ferait « en série », donc en davantage d'itérations). Puis, on réassemble 2 à 2 les listes, successivement jusqu'à retrouver une liste de même taille que la liste d'entrée.



Inspiré du cours IPT lycée Lafayette

Vous découvrirez le code associé dans le TD 22, où nous chercherons à décrire cet algorithme par **récursivité**, sa forme la plus courante. Il s'agit d'un algorithme de tri **stable**, pensé par John von Neumann en 1945.

L'algorithme **fusion** a une complexité en $O(N)$ (N étant le nombre d'éléments de la liste de sortie), et elle est appelée à chaque itération de la fonction récursive, de sorte que **tri_fusion** a une complexité en $O(N\log(N))$, que ce soit dans le meilleur ou le pire des cas (donc en moyenne également). Ce résultat est à connaître par cœur mais ne sera pas démontré en cours.

Son inconvénient majeur est qu'il n'est pas en place, et consomme donc de la ressource en RAM.

C.2 / Tri *RAPIDE* (ou *QUICKSORT*)

L'algorithme récursif de tri rapide peut s'effectuer en place ou non. Le principe d'une itération est le suivant :

1. On choisit arbitrairement un élément de la liste (par exemple le 1er, le dernier, ou un tiré au hasard).

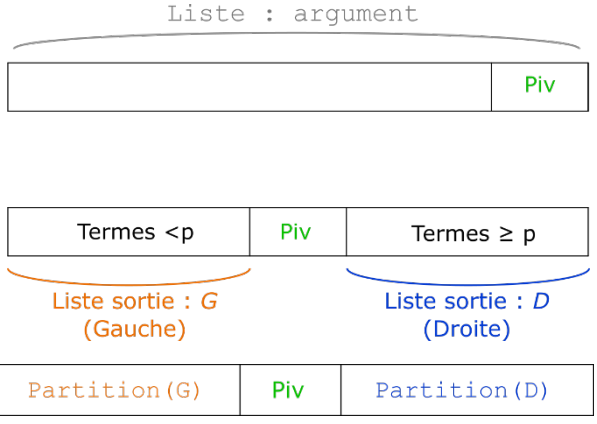
   → cet élément est appelé pivot .

   Par la suite, nous choisirons arbitrairement qu'il s'agit du dernier élément de la liste.

2. Par comparaisons successives de tous les éléments de la liste, nous déplaçons en ordre indistinct :
   a. Tous les éléments `L[i] < pivot` à gauche du pivot
   b. Tous les éléments `L[i] ≥ pivot` à droite du pivot

Ces deux étapes seront réalisées par une même fonction **`partition`**`(Liste)`, qui prend en argument une liste quelconque et renvoie, par exemple :

- Une liste *G* : correspondant aux termes à gauche du pivot
- La valeur du pivot *Piv*
- Une liste *D* : correspondant aux termes à droite du pivot.



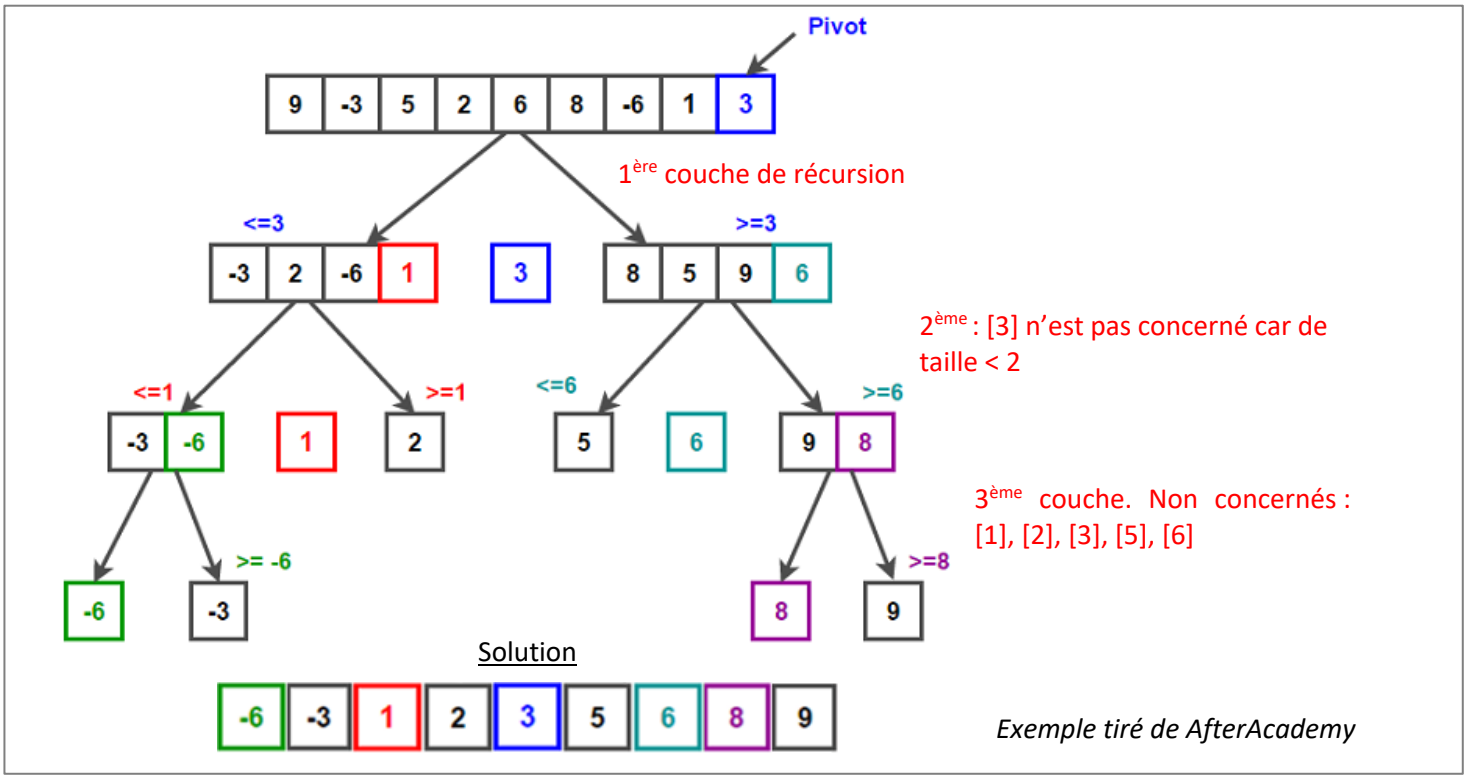
Exemples : `partition( [1,-2,0, 5,0] )` renvoie :

`[1, -2], 0, [0, 5]`

`partition( [1] )` renvoie : `[] , 1, []`

L'algorithme de tri rapide va itérer récursivement cette fonction, sur la liste de gauche et sur la liste de droite du pivot simultanément.

Afin d'assurer la terminaison de l'algorithme récursif (pas de boucle infinie), on mettra une condition d'arrêt que la fonction récursive **tri_rapide** n'effectue une **partition** que sur des listes contenant au moins 2 éléments. En-dessous de cette limite, **tri_rapide** renvoie la liste non altérée.



Vous découvrirez le code associé dans le TD 22. Il s'agit d'un algorithme de tri **instable**, pensé par C.A.R Hoare en 1961.

Cet algorithme a une complexité $O(N\log(N))$ dans le meilleur des cas et en moyenne, mais une complexité $O(N^2)$ dans le pire des cas. Cependant, le pire des cas est très peu probable. Il est toutefois possible de contourner ce risque en choisissant le point de pivot de façon aléatoire. Il est possible de l'écrire **en place**.

Expérimentalement, il est souvent légèrement supérieur au tri_fusion. Il s'agit encore aujourd'hui de la méthode de tri la plus utilisée au Monde. Toutefois, notons que l'algorithme de tri *Timsort* (hors programme CPGE), hybride entre les deux algorithmes précédents, mis au point par Tim Peters en 2002 pour *Python*, cumule les avantages des deux méthodes : stable, rapide et en place, et est légèrement plus efficace que ces deux méthodes.
La méthode `L.sort()` (applicable à un objet *L* de classe liste) ou la fonction `sorted(L)` (applicable à un argument de type liste) s'appuient toutes les deux sur l'algorithme *Timsort*.