Chapitre # (ALGO) 3 Tris Itératifs

La question de savoir si les machines peuvent penser... est à peu près aussi pertinente que celle de savoir si les sous-marins peuvent nager.

— **Edsger Dijkstra**

Résumé & Plan

L'objectif de ce chapitre est de parcourir les principaux principes de tris de listes au programme de BCPST. Nous analyserons plus tard leur complexité temporelle, de manière empirique, à l'aide du module `time` de Python.

- Les chapitres d'Informatique sont composés de cours et d'exercices intégrés. Le cours sera projeté au tableau.
- Il n'est pas attendu que toute la classe aborde tous les exercices. Traitez donc en priorité les exercices présents dans la liste donnée à chaque début de séance.
- Exercices 🌓 / **Pour aller plus loin** : exercices plus difficiles, ou plus techniques. À ne regarder que si les autres sont bien compris.

1. GÉNÉRALITÉS

L'étude des tris est une partie incontournable de l'algorithmique. En effet, dans de nombreux contextes le tri de données parait indispensable. Par exemple pour :

- trouver les 10 plus grands éléments d'une liste (par exemple, une grande liste comportant des relevés de températures sur 1 siècle, quelles sont les 10 années les plus chaudes?). Pour cette problématique, il est bien plus pratique de trier plutôt que de faire cette recherche de maximums à la main (trouver les deux plus grands éléments d'une liste n'est déjà pas si facile...).
- Calculer le minimum et le maximum d'une liste, on extrait alors les éléments de tête et de queue de la version triée,
- trouver la médiane d'une série statistique, ...on a déjà besoin de trier les données avant de pouvoir la calculer.

Trier a bien entendu un coût mais qui est vite compensé lorsque le nombre de recherches devient conséquent. Il est donc nécessaire de pouvoir trier efficacement. Nous comparerons leur efficacité avec le module `time`, donc de manière empirique. Pour commencer, mettons en place un petit test afin de savoir si une liste est triée ou non.

Exercice 1 | Une liste est-elle triée? Solution Pour savoir si une liste est triée dans l'ordre croissant, on peut :

- parcourir les éléments de la liste deux à deux,
- si une paire d'éléments consécutifs n'est pas rangée dans le bon ordre, on retournera `False`.

À la fin du parcours, si aucune paire d'éléments consécutifs était mal triée, on retourne `True`.

1. Écrire une fonction d'en-tête `tri_test_croissant(L)`, qui prend en paramètre une liste `L` et qui renvoie `True` si elle est triée par ordre croissant, et `False` sinon, selon le principe exposé précédemment.
2. Même question mais dans l'ordre décroissant.

Vocabulaire des tris. Soit `L` une liste.

- On appelle *tri* toute procédure algorithmique permettant de retourner une nouvelle liste étant une version triée de `L`, ou bien de modifier la liste `L` afin qu'elle soit triée.
- Un tri est qualifié de *tri récursif* s'il est codé récursivement (voir Chapitre (ALGO) 4), itératif sinon.

- Un tri est dit qualifié de *comparatif* s'il effectue des comparaisons entre les éléments de la liste à trier.
- Lorsque la procédure précédente modifie directement la liste ou chaîne d'entrée, on dit que le tri s'effectue *en place*. Il est dit *non en place* dans le cas contraire (création d'une nouvelle liste).

L'immense majorité des tris sont comparatifs. Nous donnerons un unique exemple de tri non-comparatif, le tri par comptage, qui n'a d'intérêt que dans une situation très particulière.

FONCTIONS DE TRI EXISTANTES. Python sait déjà trier des listes (fonction `sorted` et méthode `.sort()`), mais elles sont à éviter aux concours *(sauf si on vous l'autorise explicitement dans un sujet!)* car vous devez avant tout connaître les tris du programme.

```
>>> L = [1, 3, 2, 7, 0]
>>> L.sort() # modifie L (en place)
>>> L
[0, 1, 2, 3, 7]
>>> M = [1, 3, 2, 7, 0]
>>> sorted(M) # création d'une nouvelle liste (non en place)
[0, 1, 2, 3, 7]
```

2. TRIS ITÉRATIFS

2.1. Tri stupide *(bogosort)*

LE PRINCIPE : ON MÉLANGE JUSQU'À L'OBTENTION D'UNE LISTE TRIÉE. Le tri stupide consiste à regarder si une liste `L` donnée est triée. Dans le cas contraire, on mélangera aléatoirement ses éléments jusqu'à obtenir une liste triée.

Exercice 2 | **Programmation du tri stupide** Solution Écrire une fonction d'en-tête `tri_stupide(L)` mettant en oeuvre cet algorithme. *On pourra se servir de la fonction `shuffle` du module `random` qui mélange les éléments d'une liste, le tout en place. Voici ci-dessous une démonstration.*

```
>>> import random as rd
>>> L = [1, 3, 2]
>>> rd.shuffle(L)
>>> L
[1, 3, 2]
```

Ce tri a-t-il lieu en place? est-il comparatif? Comment faire pour obtenir une liste triée par ordre décroissant?

On se doute bien que ce tri sera très inefficace, nous le constaterons plus tard. Passons à présent aux tris itératifs « plus intelligents ». Maintenant, commençons à réfléchir un peu pour trier de manière efficace.

2.2. Tri par sélection (du minimum, version en place)

Exercice 3 | **Préliminaire : Minimum et premier indice** Solution Écrire une fonction d'en-tête `minimum_premind(L)` qui retourne le minium de `L` et le premier indice où le minimum de `L` est présent.

LE PRINCIPE : RECHERCHER LE `min` ET LE PLACER AU DÉBUT. Pour le tri par sélection, on va chercher le plus petit élément de `L` et le placer au début en permutant avec le premier élément. On recommence ensuite avec la liste `L[1:]` (liste privée de son premier élément). *(On pourrait aussi chercher le maximum, puis le placer à la fin)*

> **Exemple 1** **(Tri pas sélection (version en place))** On considère la liste `L = [1, -4, 7, 4, 2]`. Trions à la main, selon le tri par sélection, la liste `L`, en précisant bien étape par étape le minimum trouvé, et dans quelle sous-liste on l'a cherché.

Liste	min	ind_mini *(dans la liste bleue))*	Liste après échange
[1, -4, 7, 4, 2]	-4	1	[-4, 1, 7, 4, 2]
[-4, 1, 7, 4, 2]	1	0	[-4, 1, 7, 4, 2]
[-4, 1, 7, 4, 2]	2	2	[-4, 1, 2, 4, 7]
[-4, 1, 2, 4, 7]	4	0	[-4, 1, 2, 4, 7]
L'algorithme se termine en `len(L)-1` étapes.			

Exemple 2 **(Tri pas sélection (version en place))** On considère la liste `L = [3, 0, 0, 5 , 7, 5, 5]`. Trions à la main, selon le tri par sélection, la liste `L`, comme précédemment.

Exercice 4 | **Programmation du tri par sélection** Solution

1. Écrire et compléter une fonction `tri_selection` qui trie une liste `L` selon le principe du tri par sélection du minimum.

```
def tri_selection(L):
    """
    Trie la liste L selon le tri par sélection (du min) (en \
    ↪ place)
    """
    for i in range(__________):
        mini, ind_mini = minimum_premind(__________)
        # On le place au début de L[i:]
        L[___], L[__________] = L[__________], L[___]
```

2. Ce tri a-t-il lieu en place? Est-il comparatif? Comment faire pour obtenir une liste triée par ordre décroissant?

2.3. Tri par insertion

LE PRINCIPE : LE TRI D'UN JEU DE CARTE. C'est le tri du joueur de cartes. On fait comme si les éléments à trier étaient donnés un par un, le premier élément constituant, à lui tout seul, une liste triée de longueur 1. On range ensuite le second élément pour constituer une liste triée de longueur 2, puis on range le troisième élément pour avoir une liste triée de longueur 3 et ainsi de suite...

Exemple 3 **(Tri par insertion (version non en place))** On considère la liste `L = [1, -4, 7, 4, 2]`. Trions à la main, selon le tri par insertion, la liste `L`, en précisant bien étape par étape l'évolution de la version triée de `L` notée `L_tri`, et la liste `L`. On surligne à gauche l'élément qu'on souhaite insérer à droite.

Liste initiale L	Liste triée L_tri	Commentaire
[1, -4, 7, 4, 2]	[1]	
[1, -4, 7, 4, 2]	[-4, 1]	On insère -4
[1, -4, 7, 4, 2]	[-4, 1, 7]	On insère 7
[1, -4, 7, 4, 2]	[-4, 1, 4, 7]	On insère 4
[1, -4, 7, 4, 2]	[-4, 1, 2, 4, 7]	On insère 2
L'algorithme se termine en `len(L)` étapes.		

Exemple 4 **(Tri par insertion (version non en place))** On considère la liste `L = [3, 2, 0, 1, 7, 5]`. Trions à la main, selon le tri par insertion la liste `L`.

L'essentiel du travail réside donc dans l'étape d'insertion à la bonne place. Pour l'insertion en tant que telle, on s'autorisera l'utilisation d'`insert`.

Insérer un élément dans une liste (rappel)

```
>>> L = [1, 2, 3, 4]
>>> L.insert(1, -1) # l'indice d'insertion est le premier \
↪ argument
>>> L
[1, -1, 2, 3, 4]
```

Nous avons donc inséré l'élément `-1` à **gauche** de l'ancien élément d'indice `1`.

Exercice 5 | Étape d'insertion Solution Écrire, en complétant le code ci-dessous, une fonction `insertion(L_tri, x)` qui réalise l'insertion de `x` dans `L_tri` (supposée triée dans l'ordre croissant) afin que `L_tri` soit encore triée dans l'ordre croissant.

```
def insertion(L_tri, x):
    i = 0
    while (i < ______) and (______ < ______):
        i += 1
    # insertion a la bonne place
    L_tri.insert(_________, x)
```

Exercice 6 | Programmation du tri par insertion Solution

1. Créer une fonction `tri_insertion` qui trie une liste `L` selon le principe du tri par insertion.
2. Ce tri a-t-il lieu en place ? Est-il comparatif ? Comment faire pour obtenir une liste triée par ordre décroissant ?

LE PRINCIPE : RECENSER LES ÉLÉMENTS PRÉSENTS ET RECONSTITUER LA LISTE TRIÉE. On cherche à trier une liste `L` d'entiers naturels. On peut supposer que tous les éléments de la liste sont dans $[\![0, N]\!]$ où `N = max(L)`.

Le tri par comptage consiste à compter le nombre d'occurences de chaque valeur i comprise entre 0 et N et à reconstituer la liste de tous les 0 ainsi obtenus puis de tous les 1,..., de tous les N (« tous » pouvant signifier « aucun »).

Les nombres d'occurences seront stockés dans une liste de taille N+1. On l'appellera la *liste des effectifs*.

Exemple 5 (Tri par comptage (version non en place)) On considère la liste `L = [3, 0, 0, 5, 7, 5, 5]`.

- Liste des effectifs : ici la plus grande valeur de `L` est N = 7 donc on initialise la liste d'effectifs à `L_eff = [0, 0, 0, 0, 0, 0, 0, 0]` qu'on va remplir en parcourant L.
  1. `L[0] = 3` donc `L_eff = [0, 0, 0, 1, 0, 0, 0, 0]`
  2. `L[1] = 0` donc `L_eff = [1, 0, 0, 1, 0, 0, 0, 0]`
  3. `L[2] = 0` donc `L_eff = [2, 0, 0, 1, 0, 0, 0, 0]`
  4. `L[3] = 5` donc `L_eff = [2, 0, 0, 1, 0, 1, 0, 0]`
  5. `L[4] = 7` donc `L_eff = [2, 0, 0, 1, 0, 1, 0, 1]`
  6. `L[5] = 5` donc `L_eff = [2, 0, 0, 1, 0, 2, 0, 1]`
  7. `L[6] = 5` donc `L_eff = [2, 0, 0, 1, 0, 3, 0, 1]`.

  En conclusion : 2 fois le `0`, 0 fois le `1`, 0 fois le `2`, 1 fois le `3`, 0 fois le `4`, 3 fois le `5`, 0 fois le `6`, 1 fois le `7`.
- On parcourt enfin la liste des effectifs et on copie dans une nouvelle liste `L_tri` autant de fois une valeur qu'elle apparait dans la liste d'effectifs. On obtient la liste triée `[0, 0, 3, 5, 5, 5, 7]`.

Exemple 6 Préciser l'entier `N` et la liste des effectifs associée à `L = [0, 3, 1, 4, 3, 2, 5, 1, 0]`.

Exercice 7 | Tri par comptage (*counting sort*) Solution

1. Écrire une fonction d'en-tête `comptage(L)` renvoyant une liste `L_eff` dont le k-ième élément ($k \in [\![0, N]\!]$), désigne le nombre d'occurences de l'entier k dans la liste `L`. *La fonction ne devra contenir qu'une seule boucle* **`for`**.
2. En déduire une fonction d'en-tête `tri_comptage(L)`, d'argument `L`, renvoyant la liste `L` triée dans l'ordre croissant. Ce tri a-t-il lieu en place? Est-il comparatif?
3. Tester la fonction sur une liste de 20 entiers inférieurs ou égaux à 5, tirés aléatoirement à l'aide de la commande :

```
>>> import random as rd
>>> L = [rd.randint(0, 5) for _ in range(20)]
```

3. AUTRES TRIS

Cette section n'est à aborder que si tout le reste a été terminé. Les tris qui suivent sont [H.P] , et doivent être travaillés comme des exercices.

3.1. Tri à bulles

LE PRINCIPE : PARCOURS SUCCESSIFS DE LA LISTE ET ÉCHANGES DES PAIRES MAL ORDONNÉES. Le tri à *bulles* ou tri *par propagation* est un algorithme qui consiste à comparer répétitivement les paires d'éléments consécutifs d'une liste, et à les permuter lorsqu'ils sont mal triés. Il doit son nom au fait qu'il déplace rapidement les plus grands éléments en fin de tableau, comme des bulles d'air qui remonteraient rapidement à la surface d'un liquide.

Le tri à bulles est souvent enseigné en tant qu'exemple algorithmique, car son principe est simple, mais c'est le plus lent des algorithmes de tri communément enseignés, et il n'est donc guère utilisé en pratique. Un point clef est l'introduction d'un booléen qui va vérifier si après chaque parcours de liste on a eu besoin de « remonter » un élément : l'algorithme s'arrête alors lorsque plus aucun élément n'a eu besoin d'être remonté.

Exercice 8 | **Tri à bulles** Solution On considère le code à trous suivant :

```
def tri_bulles(L):
    """
    Modifie la liste L pour la trier, selon le tri à bulles
    Tri en place
    """
    echange_fait = True
    while echange_fait == True:
        echange_fait = False
        for j in range(0, len(L)-1):
            if _______ > _______:
                L[j], L[j+1] = L[j+1], L[j]
                echange_fait = _______________
```

Recopier et compléter le script ci-dessus pour qu'il corresponde au tri à bulles. Ce tri a-t-il lieu en place? Est-il itératif? récursif? comparatif? Comment faire pour obtenir une liste triée par ordre décroissant?

Exemple 7 On considère la liste `L = [1, -4, 7, 4, 2]`. Trions à la main, selon le tri bulles, la liste `L`, en précisant bien étape par étape l'état d'`echange_fait`.

Liste initiale	echange_fait	Commentaire
`L = [1, -4, 7, 4, 2]`	`True`	Début du parcours de `L`
`L = [-4, 1, 4, 2, 7]`	`True`	`[1, -4]`, `[7, 4]`, `[7, 2]` mal triées
`L = [-4, 1, 2, 4, 7]`	`True`	`[4, 2]` mal triées
`L = [-4, 1, 2, 4, 7]`	`False`	aucune paire mal triée

L'algorithme se termine, `echange_fait = False`. On constate que le tri a lieu en place puisque les permutations sont faites directement dans la liste initiale.

3.2. Tri par paquets

Exercice 9 | **Tri par paquets (*bucket sort*)** Solution On suppose que tous les éléments de la liste `L` à trier sont dans l'intervalle $I = [a, b[$, avec $a < b$ deux réels.

Le tri par paquets consiste à découper l'intervalle I en `len (L)` sous-intervalles de même longueur puis à répartir les données en paquets correspondant à chacun de ces sous-intervalles. On triera alors chacun de ces paquets à l'aide d'un des algorithmes de tri précédemment implémentés. *Vous pourrez vous servir de la fonction `sorted` dans cet exercice*

On peut montrer que ce tri est toujours plus efficace que le tri auxiliaire utilisé. En effet, dans le pire des cas tous les éléments sont dans un seul sous-intervalle, et donc cela revient à n'utiliser que le tri auxiliaire. On obtient donc un tri toujours au moins aussi efficace que l'autre.

Notons `h = (b-a)/len(L)`, `n = len(L)` et $I_0, \dots, I_{n-1}$ les sous-intervalles.

1. Écrire une fonction d'en-tête `calcul_interv(x, L)` qui étant donné un élément `x` de `L`, retourne l'entier `k` entre `0` et `n` dans lequel se trouve `x`
2. En déduire une fonction d'en-tête `tri_paquets(L)` implémentant l'algorithme de l'énoncé.

4. SOLUTIONS DES EXERCICES

Solution (exercice 1) Énoncé

```
def tri_test_croissant(L):
    """
    retourne True si L est triée par ordre croissant, et
    False sinon
    """
    for i in range(len(L)-1):
        if L[i+1] < L[i]:
            return False
    return True
def tri_test_decroissant(L):
    """
    retourne True si L est triée par ordre décroissant, et
    False sinon
    """
    for i in range(len(L)-1):
        if L[i+1] > L[i]:
            return False
    return True
```

Solution (exercice 2) Énoncé

```
import random as rd

def tri_test_croissant(L):
    """
    retourne vrai/faux selon que L est triee par ordre \
    ↪ croissant
    """
    for i in range(len(L)-1):
        if L[i+1] < L[i]:
            return False
    return True

def tri_stupide(L):
    """
    mélange les éléments de L aléatoirement jusqu'à obtenir \
    ↪ la version triée
    """
    while not tri_test_croissant(L):
        rd.shuffle(L)
>>> L = [3, 1, 7]
>>> tri_stupide(L)
>>> L
[1, 3, 7]
```

Ce tri a lieu en place, car `shuffle` modifie directement la liste de départ. Il est comparatif car `tri_test_croissant` réalise des comparaisons. Pour obtenir un tri dans l'ordre décroissant, on a plutôt recours à `tri_test_decroissant`.

Solution (exercice 3) Énoncé

```
def minimum_premind(L):
    """
    Retourne le minimum de L, et renvoie le premier indice où \
    ↪ il apparaît
    """
    mini = L[0]
    ind_mini = 0
    for k in range(1, len(L)):
        if L[k] < mini:
            mini = L[k]
            ind_mini = k
    return mini, ind_mini
>>> L = [-4, 1, 7, 4, 2]
>>> minimum_premind(L)
(-4, 0)
```

Solution (exercice 4) Énoncé

```
def tri_selection(L):
    """
    Trie la liste L selon le tri par sélection (du min) (en \
    ↪ place)
    """
    for i in range(len(L)-1):
        # Recherche du minimum de L[i:]
        mini, ind_mini = minimum_premind(L[i:])
        # On le place au début de L[i:]
        L[i], L[i + ind_mini] = L[i + ind_mini], L[i]
```

Le tri présenté ici est comparatif par l'intermédiaire de la fonction

`minimum_premind`. Pour trier en ordre décroissant : on peut simplement ajouter le minimum à la fin à chaque étape, ou alors remplacer la fonction `minimum_premind` par `maximum_premind` (qui retournerait le maximum de `L` et le premier indice où le maximum de `L` est présent).

```
>>> L = [3, 0, 0, 5 , 7, 5, 5]
>>> tri_selection(L)
>>> L
[0, 0, 3, 5, 5, 5, 7]
```

Solution (exercice 5) Énoncé

```
def insertion(L_tri, x):
    i = 0
    while (i < len(L_tri)) and (L_tri[i] < x):
        i += 1
    # insertion a la bonne place
    L_tri.insert(i, x)
>>> L_tri = [1, 2, 4]
>>> insertion(L_tri, 7) # insertion à la fin
>>> L_tri
[1, 2, 4, 7]
>>> L_tri = [1, 2, 4]
>>> insertion(L_tri, 3)
>>> L_tri
[1, 2, 3, 4]
```

Solution (exercice 6) Énoncé

```
def insertion(L_tri, x):
    i = 0
    while (i < len(L_tri)) and (L_tri[i] < x):
        i += 1
    # insertion a la bonne place
    L_tri.insert(i, x)

def tri_insertion(L):
    """
    Trie la liste L selon le tri par insertion (Non en place)
    """
    L_tri = [L[0]]
    for i in range(1, len(L)):
        insertion(L_tri, L[i])
    return L_tri
```

Solution (exercice 7) Énoncé

1.
```
def comptage(L):
    """
    renvoie le tableau des effectifs des entiers entre 1 \
    ↪ et N dans L
    """
    N = max(L)
    L_eff = [0 for _ in range(N+1)]
    for i in L:
        L_eff[i] += 1
    return L_eff
>>> L = [3, 0, 0, 5, 7, 5, 5]
>>> comptage(L)
[2, 0, 0, 1, 0, 3, 0, 1]
```

2. On construit la version triée de `L` en assemblant autant de copies nécessaires de chaque entier.

```
def tri_comptage(L):
    """
    renvoie le tableau des effectifs des entiers entre 1 \
    ↪ et N dans L
    """
    N = max(L)
    L_eff = comptage(L)
    L_tri = []
    for i in range(N+1):
        for _ in range(L_eff[i]):
            # eff[i] : nombre de fois que l'on doit \
            ↪ ajouter i dans L_tri
            L_tri.append(i)
    return L_tri
>>> L = [3, 0, 0, 5, 7, 5, 5]
>>> tri_comptage(L)
[0, 0, 3, 5, 5, 5, 7]
```

Ce tri est non en place et pour une fois non comparatif.

3. Testons la fonction tri sur une liste de 20 entiers inférieurs ou égaux à 5, tirés aléatoirement.

```
>>> import random as rd
>>> L = [rd.randint(0, 5) for _ in range(20)]
```

```
>>> tri_comptage(L)
[0, 0, 1, 1, 1, 2, 2, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5, 5, 5]
```

Solution (exercice 8) Énoncé

```
def tri_bulles(L):
    """
    Modifie la liste L pour la trier, selon le tri à bulles
    Tri en place
    """
    echange_fait = True
    while echange_fait == True:
        echange_fait = False
        for j in range(0, len(L)-1):
            if L[j] > L[j+1]:
                L[j], L[j+1] = L[j+1], L[j]
                echange_fait = True
>>> L = [3, 1, 5, 7, -2]
>>> tri_bulles(L)
>>> L # pour observer l'effet du tri
[-2, 1, 3, 5, 7]
```

Solution (exercice 9) Énoncé Utilisons par exemple le tri de Python pour tri auxiliaire.

```
def calcul_interv(x, L, a, b):
    """
    retourne l'entier k de l'intervalle Ik associé
    """
    k = 0
    n = len(L)
    h = (b-a)/n
    while a+h*(k+1) < x:
        k += 1
    return k

def tri_paquets(L, a, b):
    """
    retourne L triée dans l'ordre croissant
    """
    n = len(L)
    L_blocs = [[] for _ in range(n)] # liste contenant les \
    ↪ éléments regroupes en blocs
    for x in L:
        k = calcul_interv(x, L, a, b)
        L_blocs[k].append(x)
    L_tri = []
    for bloc in L_blocs:
        bloc_tri = sorted(bloc)
        for x in bloc_tri:
            L_tri.append(x)
    return L_tri
L = [3, 1, 7, 2]
>>> tri_paquets(L, 1, 7.1)
[1, 2, 3, 7]
```