# TP 6 : Algorithmes de tri

L'objectif de ce TP est de proposer et comparer des algorithmes permettant de trier dans l'ordre croissant les éléments d'une liste **Python**.

## I. Premiers algorithmes de tris

### I.1. Tri par sélection

Cet algorithme consiste à :

- sélectionner un élément minimal de la liste,
- échanger cet élément avec le premier élément de la liste,
- recommencer avec les éléments suivants.

▶ Trier à la main la liste `[7, 3, 2, 5, 1]` à l'aide de cet algorithme.

- Étape 0 : `[7, 3, 2, 5, 1]`
  *(rien ne se passe, on sélectionne le plus petit élément de cette liste :* 1*)*
- Étape 1 : `[1, 3, 2, 5, 7]`
  *(on échange* 1 *et* 7*, puis on sélectionne le plus petit élément de la liste* `[3, 2, 5, 7]` *:* 2*)*
- Étape 2 : `[1, 2, 3, 5, 7]`
  *(on échange* 2 *et* 3*, puis on sélectionne le plus petit élément de la liste* `[3, 5, 7]` *:* 3*)*
- Étape 3 : `[1, 2, 3, 5, 7]`
  *(rien ne se passe car* 3 *est bien placé, puis on sélectionne le plus petit élément de la liste* `[5, 7]` *:* 5*)*
- Étape 4 : `[1, 2, 3, 5, 7]`
  *(rien ne se passe car* 5 *est bien placé, puis on sélectionne le plus petit élément de la liste* `[7]` *:* 7*)*
- Étape 5 : `[1, 2, 3, 5, 7]`
  *(rien ne se passe car* 7 *est le dernier élément de la liste)*

▶ Écrire une fonction `indice_min` qui prend en paramètre une liste `L` et un entier `i`, et qui renvoie l'indice du plus petit élément de la liste `L[i:]`.
En cas d'égalité, on renverra l'indice minimal.

```
1  def indice_min(L, i) :
2      n = len(L)
3      i_min = i
4      for j in range(i, n) :
5          if L[j] < L[i_min] :
6              i_min = j
7      return i_min
```

▶ Écrire une fonction `echange(L,i,j)` qui échange les éléments d'indices `i` et `j` de la liste `L`.

```
def echange(L, i, j) :
    L[i], L[j] = L[j], L[i]
```

▶ En déduire une fonction `tri_selection` qui :

× prend en paramètre une liste `L`,

× modifie cette liste à l'aide d'une structure itérative de manière à préserver l'invariant suivant à chaque tour de boucle `i` :

***a)*** $\texttt{L[0]} \leqslant \texttt{L[1]} \leqslant \cdots \leqslant \texttt{L[i]}$

***b)*** les autres éléments sont supérieurs ou égaux à `L[i]`.

× renvoie None. On dit que cette fonction agit par **effet de bord** ou qu'elle modifie la liste `L` **en place**.

```
def tri_selection(L) :
    n = len(L)
    for i in range(n) :
        p = indice_min(L, i)
        echange(L, i, p)
    return None
```

▶ Quelle est la complexité dans le pire cas de cet algorithme ?

- On effectue toujours $n$ étapes (ce sont les $n$ tours de boucle).
- À l'étape $i$ on effectue toujours $n-i$ comparaisons pour déterminer le minimum de la liste `L[i:]`.
- On effetue alors en tout : $\displaystyle\sum_{i=0}^{n-1}(n-i) = \sum_{k=1}^{n} k = \frac{n(n+1)}{2}$ comparaisons.

## I.2. Tri par insertion

C'est la forme de tri que l'on utilise naturellement pour trier sa main lorsque l'on joue aux cartes. L'idée est de trier sa main de gauche à droite en insérant successivement au bon endroit une carte de droite dans la partie de gauche. Plus précisément, on procède comme suit.

- Initialement, la partie gauche contient 1 élément (elle est donc triée) : la carte la plus à gauche.
- La carte suivante de la main est alors placée soit avant la carte précédente, soit après.
- On procède ainsi de suite de sorte qu'au début de la $i^{\text{ème}}$ étape, les $i$ cartes les plus à gauche sont classées dans l'ordre croissant. On place alors la $i^{\text{ème}}$ carte au bon endroit dans cette partie de gauche classée.
- La dernière étape consiste à placer la carte la plus à droite parmi les $n-1$ cartes les plus à gauches qui sont alors classées correctement.

Ce tri peut être codé comme suit.

```
1  def tri_insertion(L) :
2      n = len(L)
3      for i in range(1,n):
4          aux = L[i]
5          j = i
6          while (j > 0 and L[j-1] > aux) :
7              L[j] = L[j-1]
8              j = j - 1
9          L[j] = aux
```

▶ Trier à la main la liste `[2, 5, 3, 7, 1]` à l'aide de cet algorithme.

- $1^{\text{ère}}$ étape : `[2, 5, 3, 7, 1]`.
  *(pas une vraie étape : la partie de gauche contient un seul nombre et est donc triée)*
- $2^{\text{ème}}$ étape : `[2, 5, 3, 7, 1]`.
  *(rien ne se produit : 5 est placé par rapport à 2)*
- $3^{\text{ème}}$ étape : `[2, 3, 5, 7, 1]`.
  *(3 est placé par rapport à $[2,5]$)*
- $4^{\text{ème}}$ étape : `[2, 3, 5, 7, 1]`.
  *(7 est placé par rapport à $[2,3,5]$ : rien ne se produit)*
- $5^{\text{ème}}$ étape : `[1, 2, 3, 5, 7]`.
  *(A est placé par rapport à $[2,3,5,7]$)*

▶ Quelle est la complexité dans le pire cas de cet algorithme ?

- On effectue toujours $n-1$ étapes (rien n'est fait lorsque la partie gauche ne contient qu'un élément).
- À l'étape $i$ on effectue au maximum $i$ comparaisons. Le pire cas se produit lorsque la liste apparaît initialement dans l'ordre décroissant.
- On effetue alors en tout : $\sum_{i=1}^{n-1} i = \dfrac{(n-1)\,n}{2}$ comparaisons.

▶ Que renvoie cette fonction ?

- La fonction ne contient pas d'instruction `return`. Dans ce cas, l'objet `None` (objet vide) est renvoyé. C'est la valeur par défaut lorsque la fonction ne renvoie « rien ».
- Il faut bien comprendre que la liste est triée **sur place**.
- Pour trier une liste, on commence par la définir (`L = [2, 5, 3, 7, 1]`), on réalise alors l'appel `tri_insertion(L)`. La liste `L` a été modifiée et est alors triée.

## I.3. Tri à bulles

- Cet algorithme est basé sur un mécanisme de remontée permettant, par échange de deux valeurs successives, de placer le plus grand élément en fin de liste.
  Plus précisément, la remontée s'effectue comme suit.

  On considère $a$ le premier élément de la liste,
  - × s'il est plus grand que l'élément $b$ suivant, on échange ces deux éléments, et on continue la remontée avec $a$.
  - × sinon, on n'éffectue pas d'échange et on continue la remontée avec $b$.
- L'algorithme consiste alors à effectuer $n$ remontées successives.
- Ce tri tire son nom de ce mécanisme de remontée que l'on peut rapprocher du parcours d'une bulle qui remonte progressivement à la surface d'un verre contenant une boisson gazeuse.

▶ Trier à la main la liste `[2, 5, 3, 7, 1]` à l'aide de cet algorithme.

- Lors de la 1ère remontée, la liste évolue comme suit :
  `[2,5,3,7,1]`, `[2,5,3,7,1]`, `[2,3,5,7,1]`, `[2,3,5,7,1]`, `[2,3,5,1,7]`.
  *(le plus grand élément est correctement placé)*
- Lors de la 2ème remontée, la liste évolue comme suit :
  `[2,3,5,1,7]`, `[2,3,5,1,7]`, `[2,3,5,1,7]`, `[2,3,1,5,7]`.
  *(une étape de moins est nécessaire pour placer le second plus grand élément)*
- Lors de la 3ème remontée, la liste évolue comme suit :
  `[2,3,1,5,7]`, `[2,3,1,5,7]`, `[2,1,3,5,7]`.
  *(une étape de moins est nécessaire pour placer le troisième plus grand élément)*
- Lors de la 4ème remontée, la liste évolue comme suit :
  `[2,1,3,5,7]`, `[1,2,3,5,7]`.
  *(une étape de moins est nécessaire pour placer le quatrième plus grand élément)*

Les 4 plus grands éléments de la liste étant bien placés, il en est de même du 5ème plus grand élément, qui n'est autre que le plus petit.

On propose alors le code suivant suivant.

```
1  def tri_bulles(L) :
2      n = len(L)
3      for i in range(n-1, 0, -1) :
4          nb_echange = 0
5          for j in range(i) :
6              if L[j] > L[j+1] :
7                  echange(L, j, j + 1)
8                  nb_echange = nb_echange + 1
9          if nb_echange == 0 :
10             break
```

▶ Combien de remontées successives doit-on faire ? On détaillera notamment la ligne 3.

- Si la liste de départ est de taille `n`, on doit faire en tout **au maximum** `n-1` remontées successives. En effet, La $i^{\text{ème}}$ remontée permet de positionner le $i^{\text{ème}}$ plus grand élément de la liste. Ainsi, après `n-1` remontées, les `n-1` plus grands éléments de la liste sont placées et donc le $\texttt{n}^{\text{ème}}$ aussi.
- L'instruction `for i in range(n-1, 0, -1)` permet de faire évoluer `i` de `n-1` à `1`, ce qui est en accord avec les `n-1` remontées successives évoquées précédemment.

▶ Que réalise la boucle en ligne 5 ?

Elle permet de réaliser la $\texttt{i}^{\text{ème}}$ remontée. À la fin de la $\texttt{i}^{\text{ème}}$ remontée, on a placé les `i` plus grands éléments de la liste. C'est ce qui explique le `range(i)` : on ne cherche pas à faire remonter l'élément dans les `i` dernières case de la liste.

▶ Quel est l'intérêt de l'instruction `break` dans ce code ?
Quel est l'intérêt de cette instruction de manière générale ?

- Dans le code présenté ci-dessus, on repère si la dernière remontée en date a réellement permis de faire remonter un élément. Si ce n'est pas le cas (*i.e.* s'il n'y a pas eu d'échange), c'est que la liste est alors triée.
  On peut donc interrompre le processus de remontées successives : il n'y a pas nécessité de toujours faire `n-1` remontées successives.
- De manière générale, l'instruction `break` permet d'interrompre une boucle (`while` ou `for`). Le code précédent illustre une utilisation classique :
  - × naturellement, on devrait opter pour une boucle `while` puisque l'on ne connaît pas par avance le nombre de remontées que l'on devra effectuer.
  - × on opte cependant pour une boucle `for` : cela permet de ne pas avoir à gérer un compteur ou un booléen à mettre à jour pour savoir quand on sort de la boucle (ce qui rend parfois le code un peu difficile à lire).
  - × on ajoute alors dans le code une structure conditionnelle (`if`) dont un des cas contient l'instruction `break` qui signale l'interruption de la boucle.

▶ Quel est la complexité dans le pire cas de cet algorithme ?

- On effectue au maximum $n-1$ étapes.
- À l'étape $i$ on effectue toujours $i$ comparaisons.
- Le pire cas se produit lorsque la liste apparaît initialement dans l'ordre décroissant. Lors de la $i^{\text{ème}}$ étape, le premier élément de la liste remonte à l'indice $n-i$.
- On effetue alors en tout : $\displaystyle\sum_{i=1}^{n-1}(n-i)=\sum_{i=1}^{n-1}i=\frac{(n-1)\,n}{2}$ comparaisons.

# II. Tri fusion et tri rapide

## II.1. Diviser pour régner

On emploie le terme **diviser pour régner** pour désigner l'approche algorithmique consistant à :

*1)* **Diviser** : découper un problème initial en sous-problème.

*2)* **Régner** : résoudre les sous-problèmes.
(de manière récursive -*cf* TP précédent- ou directement s'ils sont assez petits)

*3)* **Combiner** : la solution du problème initiale est obtenue en combinant les solutions des sous-problèmes.

## II.2. Tri fusion

Le tri fusion est un algorithme récursif basé sur le principe « diviser pour régner ».

Étant donnée une liste $L$, on procède comme suit.

*1)* **Diviser** : la liste $L$ est séparée en 2 (au milieu), à chaque appel.

*2)* **Régner** : chaque sous-liste est alors triée récursivement en utilisant le principe de découpage précédent. Cas initial : une liste qui ne contient qu'un élément est triée.

*3)* **Combiner** : les deux sous-listes triées sont alors combinées de sorte à ce que la liste résultat soit triée. Cette étape est appelée **fusion**.

▶ Implémenter la fonction `fusion` qui prend en paramètre deux listes `L1` et `L2` triées et renvoie la liste `L` qui réalise la fusion des deux listes précédentes.

```
def fusion(L1, L2) :
    # renvoie la liste issue de la fusion de L1 et L2
    L = []
    i,j = 0,0 # i parcourt L1 et j parcourt L2
    m,n = len(L1), len(L2)
    for k in range(m + n) :
        if i < m and (j==n or L1[i] <= L2[j]):
            L = L + [L1[i]]
            # ou L.append(L1[i])
            i = i + 1
        else :
            L = L + [L2[j]]
            # ou L.append(L2[j])
            j = j + 1
    return L
```

▶ Implémenter la fonction `tri_fusion` qui prend en paramètre une liste `L`, trie la liste selon le principe du tri fusion et renvoie la liste obtenue.

```
def tri_fusion(L) :
    # renvoie la liste issue du tri fusion fusion de L
    n = len(L)
    if n <= 1 :
        return L
    else :
        m = n // 2
        return fusion(tri_fusion(L[:m]), tri_fusion(L[m:]))
```

- Nous avons implémenté la fonction `fusion` de manière itérative. Écrire une fonction `fusionRec` permettant de réaliser la fusion de deux listes triées `L1` et `L2` de manière récursive.

```
def fusionRec(L1, L2) :
    if L1 == [] :
        return L2
    if L2 == [] :
        return L1
    if L1[0] < L2[0] :
        return [L1[0]] + fusionRec(L1[1:],L2)
    else :
        return [L2[0]] + fusionRec(L1,L2[1:])
```

- On cherche maintenant à évaluer la complexité (ou coût) de cet algorithme. Fixons $L$ une liste de taille $n$ et notons :
  - $C(n)$ la complexité en nombre d'opérations élémentaires du tri fusion.
  - $\alpha_n$ la complexité en nombre d'opérations élémentaires de l'opération de fusion de deux listes triées.

  On peut démontrer :

$$\begin{cases} C(0) = 0 \\ \forall n \in \mathbb{N},\ C(n) \ = \ C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + C\left(\left\lceil \frac{n}{2} \right\rceil\right) + \alpha_n \end{cases}$$

- Expliquer ce résultat.

► Pour déterminer $C(n)$, on se propose d'abord de réaliser l'étude de la suite $\big(V(j)\big)_{j\in\mathbb{N}}$ définie par :

$$\forall j \in \mathbb{N}, \quad V(j) = C(2^j)$$

Démontrer, pour tout $k \in \mathbb{N}^*$ : $\dfrac{V(k)}{2^k} = \displaystyle\sum_{j=1}^{k} \frac{\alpha_{2^j}}{2^j}$.

- Soit $j \in \mathbb{N}^*$.

$$\begin{aligned} & V(j) = 2\,V(j-1) + \alpha_{2^j} && \textit{(d'après 1))} \\ \text{donc} \quad & \frac{V(j)}{2^j} = \frac{V(j-1)}{2^{j-1}} + \frac{\alpha_{2^j}}{2^j} && \textit{(en divisant par } 2^j\textit{)} \\ \text{ou encore} \quad & \frac{V(j)}{2^j} - \frac{V(j-1)}{2^{j-1}} = \frac{\alpha_{2^j}}{2^j} \end{aligned}$$

- Par sommation des égalités précédentes, pour tout $k \in \mathbb{N}^*$ :

$$\begin{array}{ccc} \displaystyle\sum_{j=1}^{k} \left( \frac{V(j)}{2^j} - \frac{V(j-1)}{2^{j-1}} \right) & = & \displaystyle\sum_{j=1}^{k} \frac{\alpha_{2^j}}{2^j} \\ \| & & \\ \dfrac{V(k)}{2^k} - \cancel{\dfrac{V(0)}{2^0}} & & \end{array}$$

► On remarque ensuite : $\alpha_n = \underset{n\to+\infty}{O}(n)$. Autrement dit, il existe $(m, M) \in \mathbb{R}_+^* \times \mathbb{R}_+^*$ tel que, pour tout $n \in \mathbb{N}$ :

$$\begin{aligned} & m \times n \leqslant \alpha_n \leqslant M \times n \\ \text{donc} \quad & m \leqslant \frac{\alpha_n}{n} \leqslant M \qquad \textit{(si } n \neq 0\textit{)} \end{aligned}$$

En déduire, pour tout $k \in \mathbb{N}^*$ :

$$m \times k\,2^k \leqslant C(2^k) \leqslant M \times k\,2^k$$

Soit $k \in \mathbb{N}^*$. Par sommation :

$$\begin{array}{ccccc} \displaystyle\sum_{j=1}^{k} m & \leqslant & \displaystyle\sum_{j=1}^{k} \frac{\alpha_{2^j}}{2^j} & \leqslant & \displaystyle\sum_{j=1}^{k} M \\ \| & & \| & & \| \\ m \times k & & \dfrac{V_k}{2^k} & & M \times k \end{array}$$

Ou encore :

$$\begin{array}{ccccccc} (1) & m \times k\ 2^k & \leqslant & V_k & \leqslant & M \times k\ 2^k & (2) \\ & & & \| & & & \\ & & & C(2^k) & & & \end{array}$$

► En déduire : $\boxed{C(n) = \underset{n\to+\infty}{O}\big(n\ln(n)\big)}$.

Soit $n \in \mathbb{N}^*$. Alors il existe un unique entier $k$ tel que :

$$2^k \leqslant n < 2^{k+1} \qquad (k = \lfloor \log_2(n) \rfloor)$$

donc $C(2^k) \leqslant C(n) \leqslant C(2^{k+1})$ *(par croissance de la suite $(C_n)$)*

d'où $m \times k\, 2^k \leqslant C(n) \leqslant M \times (k+1)\, 2^{k+1}$ *(par (1) et (2))*

ainsi $m \times \lfloor \log_2(n) \rfloor\ 2^{\lfloor \log_2(n) \rfloor} \leqslant C(n) \leqslant M \times (1 + \lfloor \log_2(n) \rfloor)\ 2^{1+\lfloor \log_2(n) \rfloor}$

Finalement : $C(n) = \underset{n\to+\infty}{O}\big(n\ln(n)\big)$.

**Remarque**

- C'est une complexité plus intéressante que celle des tris par sélection ou par insertion ($\Theta(n^2)$) que vous avez peut-être déjà rencontrés. Nous verrons plus tard dans le cours qu'on ne peut pas faire mieux asymptotiquement que $\Theta(n\ln(n))$ (mais le tri fusion n'est pas le seul algorithme qui arrive à descendre à cette complexité).
- L'algorithme de tri fusion est important. Il faut savoir écrire entièrement un programme de tri fusion sans indication, et connaître sa complexité.

## II.3. Tri rapide

Le tri rapide (**Quicksort**) est un algorithme récursif lui aussi basé sur le principe « diviser pour régner ».

### II.3.a) Préliminaire : découpage de tableau

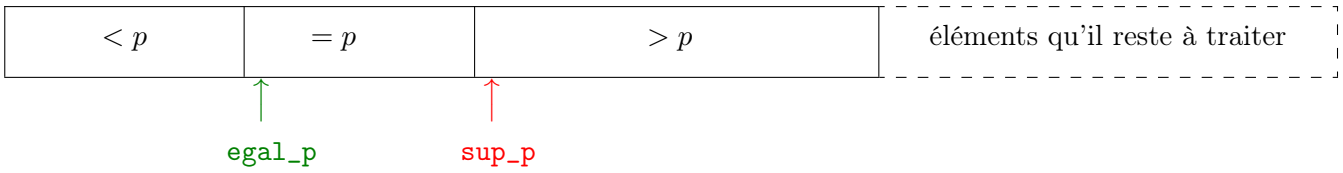
Donnons nous un tableau $T$ de taille $n$ et un élément $p$ qu'on va appeler le *pivot*. Alors il est possible de réarranger le tableau $T$ en trois parties, la première contenant les éléments strictement plus petits que $p$, la seconde les éléments égaux à $p$ et la troisième les éléments strictement plus grands que $p$.

| $< p$ | $= p$ | $> p$ |
|---|---|---|

Ce découpage peut se faire à l'aide d'un parcours du tableau et d'échanges successifs d'éléments. Plus précisément, on commence par intialiser à 0 deux variables nommées `egal_p` et `sup_p`.
Les échanges sont effectués de sorte à maintenir, pour tout $i \in [\![1, n]\!]$ la propriété $\mathscr{P}(i)$ suivante :

> Avant le $i^{\text{ème}}$ tour de boucle :
> - les $i-1$ premiers éléments du tableau initial ont été testés et sont situés dans une position d'indice inférieur ou égal à $i-1$.
> - tous les éléments testés qui sont plus petits que $p$ sont dans une position d'indice strictement inférieur à `egal_p`.
> - tous les éléments testés qui sont plus grands que $p$ sont dans une position d'indice supérieur ou égal à `sup_p`.

Ce que l'on peut représenter graphiquement par le schéma suivant :




**Remarque**

- Il est à noter que, si le pivot choisi est plus grand que le plus grand élément du tableau, alors la variable `sup_p` prend pour valeur la taille du tableau (ce qui représenterait un indice en dehors du tableau).
- S'il n'y a pas d'élement égal au pivot (ce ne peut être le cas que si l'on choisit comme pivot un élément qui n'appartient pas au tableau), alors `egal_p` = `sup_p`.

### II.3.b) Principe de l'algorithme

Étant donnée une liste $L$, on procède comme suit.

*1)* **Diviser** : on choisit le $1^{\text{er}}$ élément de $L$ comme *pivot* permettant de partitionner $L$ en deux sous-listes.
- La $1^{\text{ère}}$ contient des éléments inférieurs ou égaux au pivot.
- La $2^{\text{ème}}$ contient des éléments supérieurs (strictement) au pivot.

*2)* **Régner** : on lance l'algorithme du tri rapide sur chacune des deux sous-listes de sorte à obtenir récursivement deux sous-listes triées.

*3)* **Combiner** : le pivot est replacé au bon endroit

**Remarque**
L'algorithme du tri rapide peut être implémenté en place. Ceci signifie que l'algorithme permet de trier une liste en paramètre en modifiant directement cette liste et sans avoir besoin d'en créer une autre. Nous nous intéressons à cette implémentation.

### II.3.c) Implémentation

- Écrire une fonction `partition(L, g, d)` qui considère la liste `L[g : d+1]` et effectue la partition sur cette sous-liste. Plus précisément, les éléments de cette sous-liste devront être modifiés de sorte à ce que :
  - × les premiers éléments soient inférieurs ou égaux à `L[g]`, pivot de cette partition.
  - × les éléments suivants soient strictement supérieurs à `L[g]`.

  Le pivot sera placé au bon endroit et on renverra la position finale de cet élément.
  *(indication : on pourra utiliser un compteur* `i` *parcourant la liste de gauche à droite et un compteur* `j` *parcourant la liste de droite à gauche)*

```
def partition(L, g, d) :
    pivot = L[g]
    if g >= d :
        return g
    i = g + 1
    j = d
    while i < j :
        if L[i] <= pivot :
            i = i + 1
        elif L[j] > pivot :
            j = j - 1
        else :
            echange(L, i, j)
            i = i + 1
            j = j - 1
    if L[i] < pivot :
        echange(L, g, i)
        return i
    else :
        echange(L, g, i-1)
        return i-1
```

- Écrire alors une fonction récursive `tri_rapide(L, g, d)`.

```
def tri_rapide(L, g, d) :
    if g <= d :
        k = partition(L, g, d)
        tri_rapide(L, g, k-1)
        tri_rapide(L, k+1, d)
```

**Remarque**
La complexité du tri rapide est en moyenne (et à une constante près) de l'ordre de $n\log(n)$ comparaisons/ affectations, et de $n^2$ comparaisons/affectations dans le pire cas. On notera qu'il est possible de démontrer que l'ordre de grandeur optimal d'un tri est de $n\log(n)$ comparaisons.

# III. Tri par comptage

Supposons que les valeurs à trier sont des entiers compris entre 0 et un entier naturel $m$.
L'idée pour trier une liste comportant de telles valeurs est de dénombrer les occurences de chaque entier à l'aide d'un dictionnaire `dico_occurence` de longueur au plus $m+1$, puis de reconstruire la liste triée des valeurs à partir de ce dictionnaire.

▶ Quel appel permet de créer un dictionnaire dont les clés sont les entiers de 0 à $m$ ?

```
{ k:0 for k in range(m+1) }
```

▶ À l'aide de cet appel, proposer une fonction `tri_comptage_1` qui prend en paramètre un entier `m` et une liste `L` d'entiers de $[\![0, m]\!]$, et qui renvoie la liste `L` triée dans l'ordre croissant.

```
def tri_comptage_1(L, m) :
    L_triee = []
    dico_occurence = { k:0 for k in range(m + 1) }
    for elt in L :
        dico_occurence[elt] += 1
    for k in range(m + 1) :
        L_triee = L_triee + [k]*dico_occurence[k]
    return L_triee
```

▶ Déterminer la complexité de cet algorithme de tri.

On note $n$ la taille de la liste `L`.

- On commence par faire $n$ mise à jour du dictionnaire `dico_occurence`.
- On effectue ensuite `m` $+ 1$ mise à jour de la liste `L_triee`.
- On effectue alors en tout : $n + m + 1$ opérations.

▶ Considérons la liste `L` définie par `L = [0]*11`. Commentez l'effet de l'appel `tri_comptage_1(L, 10**9)` sur la liste `L` ainsi définie.

▶ Proposer alors une nouvelle fonction `tri_comptage_2` qui corrige ce problème.

```
def tri_comptage_2(L, m) :
    L_triee = []
    dico_occurence = {}
    for elt in L :
        if elt in dico_occurence :
            dico_occurence[elt] += 1
        else :
            dico_occurence[elt] = 1
    cles = list(dico_occurence.keys())
    cles = tri_fusion(cles)
    for k in cles :
        L_triee = L_triee + [k]*dico_occurence[k]
    return L_triee
```

▶ Quelle est la complexité de ce nouvel algorithme ?

On note $n$ la taille de la liste `L`.

- On commence par faire dans le pire des cas $n$ mise à jour du dictionnaire `dico_occurence`.
- On effectue ensuite un tri fusion de complexité dans le pire des cas $\underset{n\to+\infty}{O}\big(n\ln(n)\big)$
- On effectue alors en tout : $n + \underset{n\to+\infty}{O}\big(n\ln(n)\big) = \underset{n\to+\infty}{O}\big(n\ln(n)\big)$ opérations.