Cours_Tris_Premiere_NSI_2021_Correction

January 12, 2022

Ce fichier est un notebook Python.

Il comporte deux types de cellules :

- les cellules d'édition dans lesquelles vous pouvez saisir du texte éventuellement enrichi de mises en formes ou de liens hypertextes avec la syntaxe du langage HTML simplifié qui s'appelle Markdown. Voir http://daringfireball.net/projects/markdown/ pour la syntaxe de Markdown.
- les cellules de code où l'on peut saisir du code Python3 puis le faire exécuter avec la combinaison de touches `CTRL + RETURN`

Une cellule peut être éditée de deux façons différentes :

- en mode *commande* lorsqu'on clique sur sa marge gauche qui est surlignée alors en bleu, on peut alors :
  - changer le type de la cellule en appuyant sur `m` pour passer en cellule Markdown ou sur `y` pour passer en cellule de code
  - insérer une cellule juste au-dessus en appuyant sur `a`
  - insérer une cellule juste en-dessous en appuyant sur `b`
  - couper la cellule en appuyant sur `x` etc ...
- en mode *édition* lorsqu'on clique sur l'intérieur de la cellule.

L'aide complète sur les raccourcis claviers est accessible depuis le bouton `Help` dans la barre d'outils ci-dessus.

Exercice 1

[1]:
```
def tri_croissant(tab):
    """
    Détermine si le tableau d'entiers tab
    est trié dans l'ordre croissant

    Parameters:
        tab : tableau d'entiers
        Précondition : len(tab) > 0

    Returns:  booléen
    """
```

```
    assert len(tab) > 0
    for i in range(0, len(tab)- 1):
        if tab[i] > tab[i+1]:
            return False
    return True


#Tests unitaires
def test_tri_croissant():
    """Tests unitaires pour tri_croissant"""
    assert tri_croissant([20])
    assert tri_croissant([1, 1])
    assert tri_croissant([-2, 0, 0, 1])
    assert not tri_croissant([2, 1, 3])
    assert not tri_croissant([2, 1])
    assert not tri_croissant([2, 2, 1])
    print("Tests unitaires réussis")

# décommenter pour exécuter les tests unitaires
test_tri_croissant()
```

```
Tests unitaires réussis
```

Exercice 2

[2]:
```
import random

def melange(tab):
    """
    Mélange  en place les éléments du tableau tab

    Parameters
    ----------
    tab : tableau
    Returns
    -------
    None.

    """
    n = len(tab)
    for i in range(n):
        r = random.randint(i, n)
        tmp = tab[r]
        tab[r] = tab[i]
        tab[i] = tmp
```

Exercice 3

[3]:
```python
def compare_tab(tab1, tab2):
    """
    Détermine si les tableaux d'entiers tab1
    et tab2 triés et de même taille
    comportent les memes éléments

    Parameters: tab1, tab 2: tableaux d'entiers

    Returns: booléen
    """
    for i in range(len(tab1)):
        if tab1[i] != tab2[i]:
            return False
    return True


#jeu de tests unitaires pour une fonction de comparaison de tableaux
def test_comparaison_tableaux(fonc_compar):
    assert fonc_compar([1, 2, 3], [1, 2, 3])
    assert fonc_compar([1, 2, 2], [1, 2, 2])
    assert fonc_compar([10], [10])
    assert not fonc_compar([1, 2, 3], [1, 2, 4])
    assert not fonc_compar([10], [-4])
    assert not fonc_compar([1, 2, 3, 3], [1, 2, 3, 4])
    print("Tests unitaires réussis")

#application  des test unitaires à compare_tab (à décommenter)
test_comparaison_tableaux(compare_tab)
```

```
Tests unitaires réussis
```

Exercice 5

[16]:
```python
def recherche_index_min(t, i):
    """
    Renvoie l'index de min(t[i:len(t)])

    Parameters
    ----------
    t : tableau d'entiers
    k : int
    Précondition :   0 <= a  < len(t)
    Returns:
    -------
    int
    """
    assert 0 <= i < len(t)
    imin = i
```

```
    for j in range(i + 1, len(t)):
        if t[j] < t[imin]:
            imin = j
    return imin

def echange(t, a, b):
    """
    Echange t[a] et t[b]

    Parameters
    ----------
    t : tableau d'entiers
    a : int
    b :  int
    Précondition :  0 <= a < len(t) et 0 <= b < len(t)

    Returns
    -------
    None.

    """
    tmp = t[a]
    t[a] = t[b]
    t[b] = tmp

def tri_selection(t):
    """
    Trie en place par sélection un tableau d'entiers

    Parameters :
    t : tableau d'entiers

    Returns :  None.
    """
    n = len(t)
    #boucle externe
    for i in range(0, n):
        #invariant : t[0:i] trié dans l'ordre croissant et t[0:i] <= t[i:n]
        #boucle interne de recherche de l'index du minimum dans t[i:n]
        imin = recherche_index_min(t, i)
        #t[imin] = min(t[i:n])
        #echange de t[i] et t[imin]
        echange(t, i, imin)

import random

def test_tri(tri_en_place):
```

```
    """
    Jeu de tests unitaires pour une fonction de tri en place
    par comparaison avec la fonction de bibliothèque sorted

    Parameters
    ----------
    tri_en_place : fonction de tri  en place

    Returns
    -------
    None.

    """
    taille = [10**k for k in range(3)]
    for n in taille:
        #test sur tableau constant
        for const in [-n, 0, n]:
            t = [const for _ in range(n)]
            t_tri = sorted(t)
            tri_en_place(t)
            assert t == t_tri
        #tests sur tableaux déjà triés dans l'ordre croissant
        for debut in [-(n//2), 0, n//2]:
            t = [e for e in range(debut, debut + n)]
            t_tri = sorted(t)
            tri_en_place(t)
            assert t == t_tri
        #tests sur tableaux déjà triés dans l'ordre décroissant
        for debut in [-(n//2), 0, n//2]:
            t = [e for e in range(debut + n, debut - 1, -1)]
            t_tri = sorted(t)
            tri_en_place(t)
            assert t == t_tri
        #tests sur des tableaux aléatoires
        for _ in range(50):
            t = [random.randint(-n, n) for _ in range(n)]
            t_tri = sorted(t)
            tri_en_place(t)
            assert t == t_tri
    print(f"jeu de tests { test_tri.__name__} réussi pour {tri_en_place.
 ↪__name__}")


#tests unitaires pour le tri par sélection (à décommenter)
test_tri(tri_selection)
```

```
jeu de tests test_tri réussi pour tri_selection
```

[17]:
```python
def tri_selection2(t):
    """
    Trie en place par sélection un tableau d'entiers

    Parameters :
    t : tableau d'entiers

    Returns :  None.
    """
    n = len(t)
    #boucle externe
    for i in range(0, n):
        #invariant : t[0:i] trié et et t[0:i] <= t[i:n]
        imin = i
        for j in range(i + 1, n):
            if t[j] < t[imin]:
                imin = j
        t[imin], t[i] = t[i], t[imin]

#tests unitaires pour le tri par sélection (à décommenter)
test_tri(tri_selection2)
```

```
jeu de tests test_tri réussi pour tri_selection2
```

Exercice 8

[6]:
```python
import time, random

def temps_echantillon(tri_en_place, taille_tab, nb_essais):
    """
    Mesure le temps d'exécution de la fonction de tri tri_en_place
    sur un échantillon de nb_essais tableaux d'entiers aléatoires
    de taille taille_tab (entiers entre -1000 et 1000)

    Parameters
    ----------
    tri_en_place : fonction de tri en place
    taille_tab : int, taille de tableaux
    nb_essais : int, taille d'échantillons

    Returns
    -------
    None.

    """
    total = 0
    for _ in range(nb_essais):
        tab_alea = [random.randint(-1000, 1000) for _ in range(taille_tab)]
```

```
        debut = time.perf_counter()
        tri_en_place(tab_alea)
        total = total + (time.perf_counter() - debut)
    return total

def test_doubling_ratio(tri_en_place, nb_essais):
    """
    Description à compléter

    Parameters
    ----------
    tri_en_place : fonction de tri en place d'un tableau
    nb_essais : int

    Returns
    -------
    None.

    """
    #boucle infinie à interrompre avec le bouton Stop ou CTRL + C
    n = 128
    for _ in range(5):
          precedent = temps_echantillon(tri_en_place, n // 2,  nb_essais)
          courant = temps_echantillon(tri_en_place, n,  nb_essais)
          ratio = courant / precedent
          print(n, ratio)
          n = n * 2
```

[7]:
```
test_doubling_ratio(tri_selection, 10)
```

```
128 2.7999999999994314
256 3.448275862069025
512 5.029126213592477
1024 4.584980237154263
2048 4.226879861711322
```

```
def test_doubling_ratio(tri_en_place, nb_essais):
    """
    Affiche le ratio entre le temps moyen d'exécution de la fonction tri_en_place
    sur un échantillon de nb_essais tableaux aléatoires de taille n
    et un échantillon de nb_essais tableaux aléatoires de taille n // 2
    Double la valeur de n à chaque itération
    5 itérations (limitation pour Capytale)

    Parameters
    ----------
    tri_en_place : fonction de tri en place d'un tableau
    nb_essais : int
```

```
    Returns
    -------
    None.

    """
    #boucle infinie à interrompre avec le bouton Stop ou CTRL + C
    n = 128
    while True:
          precedent = temps_echantillon(tri_en_place, n // 2,  nb_essais)
          courant = temps_echantillon(tri_en_place, n,  nb_essais)
          ratio = courant / precedent
          print(n, ratio)
          n = n * 2
```

Exemple de test de performance de la fonction`tri_selection` :

```
>>> test_doubling_ratio(tri_selection, 10)
128 2.7999999999994314
256 3.448275862069025
512 5.029126213592477
1024 4.584980237154263
2048 4.226879861711322
```

- On observe que le temps d'exécution du *tri par sélection* est multiplié par 4 environ lorsque la taille de l'entrée est multiplié par 2.
- Le nombre de comparaisons effectué par le *tri par sélection* ne dépend pas des données à trier (déjà triées ou non).
- Si on suppose que le temps d'exécution dépend essentiellement du nombre de comparaisons effectuées, puisque le temps d'exécution quadruple lorsque la taille double, on peut conjecturer que l'ordre de grandeur du nombre de comparaisons est en n^2.
- Démontrons cette conjecture pour une entrée qui est un tableau de taille n :
  - Pour chaque tour de boucle externe d'indice i avec $0 \leqslant i \leqslant n-1$, on effectue $n-1-i$ comparaisons dans la boucle interne.
  - Au total on effectue donc :
    - $n-1$ comparaisons pour $i=0$
    - $n-2$ comparaisons pour $i=1$
    - .....
    - $n-1-j$ comparaisons pour $i=j$
    - ....
    - 0 comparaisons pour $i=n-1$
  - Soit un total de $T(n) = 0+1+2+...+(n-2)+(n-1)$ comparaisons
  - En regroupant les termes judicieusement, il vient :

$$T(n)+T(n) = (0+n-1)+(1+n-2)+...+(n-1+0)$$

  - On en déduit que :

$$T(n) = \frac{n(n-1)}{2} \approx \frac{n^2}{2}$$

  - L'ordre de grandeur du nombre de comparaisons effectuées par le tri par sélection est donc bien n^2 à une constante près. On dit que l'algorithme de tri par sélection a une **complexité quaratique**.

- En exécutant `test_doubling_ratio(tri_selection, 1)` on obtient les résultats ci-dessous. On peut observer qu'il faut environ 8 secondes pour trier un tableau de taille $2^{14} = 16384$ et environ 4 fois plus de temps pour trier un tableau de taille $2^{15} = 32768$.

Comme $10^6 \approx (2^{10})^3 = 2^{30}$, sur la même machine il faudrait environ $4^{16} \times 8 \approx 34359738368$ secondes soit environ 400000 jours pour trier un tableau de taille 10^6 avec le tri par sélection !

```
n   ratio              temps (s)
128 2.2289496220670393 0.0008636889979243279
256 3.042886778028981 0.0029914619990449864
512 5.087253289077743 0.012196878000395373
1024 3.363716977320147 0.03419660300278338
2048 3.9497047288478346 0.12215449300128967
4096 3.542892641603348 0.5020957160013495
8192 3.6109882929251853 1.9764786020023166
16384 3.860320264400124 7.705011586000182
```

Exercice 9

[15]:
```python
def insertion(t, i):
    """
    Insère t[i] à sa place dans t[:i] trié dans l'ordre croissant

    Parameters
    ----------
    t : tableau d'entiers, précondition len(t) > i
    i : int
    Returns
    -------
    None.

    """
    assert len(t) > i #précondition
    j = i
    while j > 0 and t[j] < t[j-1]:
        t[j], t[j-1] = t[j-1], t[j]
        j = j - 1


def tri_insertion(t):
    """
    Trie en place par insertion un tableau d'entiers

    Parameters :
    t : tableau d'entiers
```

```
    Returns :  None.
    """
    n = len(t)
    #boucle externe
    for i in range(1, n):
        #invariant de boucle : t[0:i] trié
        #boucle interne d'insertion de t[i] à sa place dans t[0:i]
        insertion(t, i)

#tests unitaires pour le tri par insertion (à décommenter)
test_tri(tri_insertion)
```

```
jeu de tests test_tri réussi pour tri_insertion
```

[9]:
```
def insertion2(t, i):
    """
    Insère t[i] à sa place dans t[:i] trié dans l'ordre croissant

    Parameters[
    ----------
    t : tableau d'entiers, précondition len(t) > i
    i : int
    Returns
    -------
    None.

    """
    assert len(t) > i #précondition
    tmp = t[i]
    j = i
    # à compléter
    while j > 0 and tmp < t[j-1]:
        t[j] = t[j-1]
        j = j - 1
    t[j] = tm

def tri_insertion2(t):
    """
    Trie en place par insertion un tableau d'entiers

    Parameters :
    t : tableau d'entiers

    Returns :  None.
    """
    n = len(t)
```

```
    #boucle externe
    for i in range(1, n):
        #invariant de boucle : t[0:i] trié
        #boucle interne d'insertion de t[i] à sa place dans t[0:i]
        j = i
        while j > 0 and t[j] < t[j-1]:
            t[j], t[j-1] = t[j-1], t[j]
            j = j - 1


#tests unitaires pour le tri par insertion (à décommenter)
test_tri(tri_insertion2)
```

```
jeu de tests test_tri réussi pour tri_insertion2
```

[10]:
```
test_doubling_ratio(tri_insertion, 10)
```

```
128 2.090909090927882
256 3.9183673469221083
512 4.9368421052788785
1024 3.7080745341586137
2048 4.156134686347443
```

Exercice 10

- Considérons l'évaluation de `tri_insertion(t)` avec un tableau `t` de taille n déjà trié dans l'ordre croissant :
  - Pour chaque tour de boucle externe d'indice i avec $0 \leqslant i \leqslant n-1$, la boucle interne de `insertion(t, i)` ne s'exécute pas car la condition d'entrée de boucle `t[j-1] > t[j]` n'est pas vérifiée pour $i = j$.
  - Une seule comparaison est donc effectuée dans `insertion(t, i)` pour tout indice i de boucle externe avec $0 \leqslant i \leqslant n-1$ et c'est le nombre minimal de comparaisons pour `insertion(t, i)`.
  - Au total n comparaisons sont donc effectués et c'est le nombre minimal de comparaisons possibles. La **complexité** du *tri par insertion* dans le **meilleur des cas** est donc **linéaire**.
- Au contraire, considérons l'évaluation de `tri_insertion(t)` avec un tableau `t` de taille n déjà trié dans l'ordre décroissant :
  - Pour chaque tour de boucle externe d'indice i avec $0 \leqslant i \leqslant n-1$, la boucle interne de `insertion(t, i)` s'exécute avec le nombre maximum de tours c'est-à-dire i.
  - Au total on effectue donc :
    - 0 comparaisons pour $i = 0$
    - 1 comparaisons pour $i = 1$
    - .....
    - j comparaisons pour $i = j$
    - ....
    - $n-1$ comparaisons pour $i = n-1$
  - Soit un total de $T(n) = 0 + 1 + 2 + ... + (n-2) + (n-1)$ comparaisons

– En regroupant les termes judicieusement, il vient :

$$T(n) + T(n) = (0 + n - 1) + (1 + n - 2) + ... + (n - 1 + 0)$$

– On en déduit que :

$$T(n) = \frac{n(n-1)}{2} \approx \frac{n^2}{2}$$

– L'ordre de grandeur du nombre de comparaisons effectuées par le *tri par insertion* lorsque le tableau est trié dans l'ordre inverse (ici décroissant) est donc maximal et c'est n^2 à une constante près comme pour le *tri par sélection*. La complexité du *tri par insertion* dans le **pire des cas** est donc la même que celle du *tri par sélection* dans tous les cas.

```
>>> test_doubling_ratio(tri_insertion, 10)
128 2.9362770819578694
256 4.042328118210682
512 4.483588294003115
1024 4.38177808348038
2048 4.063818466385776
```

- On observe que le temps d'exécution du *tri par insertion* est multiplié par 4 environ lorsque la taille de l'entrée est multiplié par 2 (tests sur des échantillons de tableaux aléatoires de même taille).
- Si on suppose que le temps d'exécution dépend essentiellement du nombre de comparaisons effectuées, puisque le temps d'exécution quadruple lorsque la taille double, on peut conjecturer que l'ordre de grandeur du nombre de comparaisons du tri par insertion pour un tableau quelconque (données aléatoires) est en n^2. On retrouve l'ordre de grandeur de la complexité dans le **pire des cas**.
- Comparaison *tri par sélection* / *tri par insertion* :
  - Contrairement au *tri par sélection* le nombre de comparaisons effectué par le *tri par insertion* (mesurant sa complexité temporelle) dépend des données à trier :
    * L'ordre de grandeur de la **complexité** du *tri par insertion* dans le **meilleur des cas** est **linéaire** : proportionnelle à la taille n du tableau.
    * Dans le **pire des cas** (tableau trié dans l'ordre inverse) et pour un tableau aléatoire, la **complexité** du *tri par insertion* est la même que pour le *tri par sélection* dans tous les cas, c'est-à-dire **quadratique** : proportionnelle au carré n^2 de la taille n du tableau.
  - En pratique, les cas où les données sont presque triées peuvent être assez fréquents et le **tri par insertion** se montre alors très efficace car presque **linéaire**. D'ailleurs les fonctions de tri de bibliothèques implémentées avec des algorithmes de tri qui ont u une meilleur complexité dans le pire des cas ou sur des tableaux aléatoires, peuvent basculer sur le tri par insertion pour des tableaux de petite taille ou preque triés.

Exercice 11

- Implémentations du *tri par sélection* :

En langage C :

```
void mystery_sort4(int *a, int n) {
    int i, j, m, t;
```

```
    for (i = 0; i < n; i++) {
        for (j = i, m = i; j < n; j++) {
            if (a[j] < a[m]) {
                m = j;
            }
        }
        t = a[i];
        a[i] = a[m];
        a[m] = t;
    }
```

En langage javascript :

```
function mystery_sort2(nums) {
  var len = nums.length;
  for(var i = 0; i < len; i++) {
    var minAt = i;
    for(var j = i + 1; j < len; j++) {
      if(nums[j] < nums[minAt])
        minAt = j;
    }

    if(minAt != i) {
      var temp = nums[i];
      nums[i] = nums[minAt];
      nums[minAt] = temp;
    }
  }
  return nums;
}
```

- Implémentations du *tri par insertion* :

En langage C

```
void mystery_sort1(int *a, int n) {
    for(size_t i = 1; i < n; ++i) {
        int tmp = a[i];
        size_t j = i;
        while(j > 0 && tmp < a[j - 1]) {
            a[j] = a[j - 1];
            --j;
        }
        a[j] = tmp;
    }
}
```

En langage Javascript

```
function mystery_sort3(a) {
    for (var i = 0; i < a.length; i++) {
```

```
        var k = a[i];
        for (var j = i; j > 0 && k < a[j - 1]; j--)
            a[j] = a[j - 1];
        a[j] = k;
    }
    return a;
}
```

Exercice 12

Le _tri par bulle___ (dans l'ordre croissant), est constitué de deux boucles imbriquées :

- La boucle externe s'exécute n fois
- La boucle interne balaie le tableau de gauche à droite en permutant deux éléments adjacents s'ils ne sont pas dans l'ordre croissant. Ainsi à la fin de chaque boucle interne le plus grand élément qui n'est pas encore à sa place, est remonté jusqu'à position finale.

Première amélioration :

- On n'exécute la boucle interne que sur la partie du tableau non triée c'est-à-dire les $n-1-i$ premiers éléments au tour de boucle externe d'indice i.

Seconde amélioration :

- On n'exécute la boucle externe que si deux éléments ont été permutés lors du dernier tour de boucle interne. On peut remplacer la boucle externe `for` par une boucle `while` avec comme condition un booléen positionné à `True` uniquement si on a une permutation entre deux éléments adjacents.

Complexité pour `tri_bulle3(t)` version optimisée du *tri par bulles* :

- Le *tri par bulles* est sensible aux données en entrée comme le *tri par insertion*.
- Le **meilleur des cas** est celui d'un tableau de taille n déjà trié dans le bon ordre (ici croissant), la boucle externe s'exécute une seule fois avec une boucle interne qui s'exécute $n-1$ fois.
- Le _pire des cas___ est celui d'un tableau de taille n trié dans l'ordre inverse (ici décroissant), la boucle externe s'exécute $n-1$ fois avec au total $n-1+n-2+...+1=\frac{n(n-1)}{2}$ au total soit une complexité du même ordre de grandeur que le *tri par sélection* ou le *tri par insertion* dans le **pire des cas**. Le *tri par bulles* est alors moins performant que les deux autres car le nombre d'affectations est beaucoup plus important du même ordre de grandeur que le nombre de comparaisons (n^2).

```
def tri_bulle1(tab):
    n = len(tab)
    for i in range(n):
        for j in range(n - 1):
            if tab[j] > tab[j + 1]:
                echange(tab, j, j + 1)

#tests unitaires de tri_bulle1 en place, à décommenter
#test_tri(tri_bulle1)

def tri_bulle2(tab):
```

```
    n = len(tab)
    for i in range(n):
        for j in range(n - 1 - i):
            if tab[j] > tab[j + 1]:
                echange(tab, j, j + 1)


def tri_bulle3(tab):
    n, i, permutation  = len(tab), 0, True
    while permutation:
        permutation = False
        for j in range(n - 1 - i):
            if tab[j] > tab[j + 1]:
                echange(tab, j, j + 1)
                permutation = True
        i = i + 1
```

Exercice 13

[20]:
```
def tri_comptage(tab, binf, bsup):
    """
    Tri en place par comptage le tableau
    tab tel que pour tout 0 <= k < len(tab)
    on a  binf <= tab[k] <= bsup

    Parameters
    ----------
    tab : tableau d'entiers
    binf, bsup : int et int
    Returns
    -------
    None.

    """
    histo = [0 for _ in range(bsup - binf + 1)]
    #on remplit l'histogramme
    for x in tab:
        histo[x - binf] = histo[x - binf] + 1
    #on reconstitue le tableau
    i = 0  #index dans tab
    y = 0  #index dans histo
    n = len(tab)
    while i < n:
        if histo[y] == 0:
            y = y + 1
        else:
            #y = x - binf donc x = y + binf
            tab[i] = y + binf
```

```
            histo[y] = histo[y] - 1
            i = i + 1


def test_tri_comptage(tri_en_place):
    """
    Jeu de tests unitaires pour une fonction de tri comptage
    de signature tri(tab, binf, bsup)
    Comparaison avec la fonction de tri de bibliothèque sorted

    Parameters
    ----------
    tri : fonction de tri comptage

    Returns
    -------
    None.

    """

    for n in [10**k for k in range(3)]:
        for binf, bsup in  [(-100, -1), (-100, 100), (0, 100)]:
            #test sur tableau constant
            for const in [random.randint(binf, bsup) for _ in range(5)]:
                t = [const for _ in range(n)]
                t_tri = sorted(t)
                tri_en_place(t, binf, bsup)
                assert t == t_tri
            # sur tableaux déjà triés dans l'ordre croissant
            t = [e for e in range(binf, bsup + 1)]
            t_tri = sorted(t)
            tri_en_place(t, binf, bsup)
            assert t == t_tri
            #tests sur tableaux déjà triés dans l'ordre décroissant
            t = [e for e in range(bsup, binf - 1, -1)]
            t_tri = sorted(t)
            tri_en_place(t, binf, bsup)
            assert t == t_tri
            #tests sur des tableaux aléatoires
            for _ in range(50):
                t = [random.randint(binf, bsup) for _ in range(n)]
                t_tri = sorted(t)
                tri_en_place(t, binf, bsup)
                assert t == t_tri
    print(f"jeu de tests { test_tri_comptage.__name__} réussi pour␣
↪{tri_en_place.__name__}")
```

```
# tests unitaires pour tri_comptage, à décommenter
test_tri_comptage(tri_comptage)
```

```
jeu de tests test_tri_comptage réussi pour tri_comptage
```

Le *tri par comptage* (dans l'ordre croissant), n'est pas constitué de deux boucles imbriquées comme les algorithmes de tri *quadratiques* mais de deux boucle successives :

- une boucle pour construire un histogramme des valeurs du tableau (nécessité d'utiliser un dictionnaire si l'amplitude entre le minimum et le maximum est importante)
- une boucle pour reconstituer le tableau trié à partir de l'histogramme.

Le *tri par comptage* n'est possible que si on dispose d'une informations sur les valeurs possibles du tableau ! Ce n'est malheureusement pas toujours le cas, mais cela peut correspondre à de nombreuses situations (classement des notes d'un devoir par exemple).

[23]:
```
def tri_comptage_mille(tab):
    """
aPour mesurer le temps d'exécution avec test_doubling_ratio
    qui manipule des tableaux dont les valeurs sont entre -1000
    et 1000
    """
    return tri_comptage(tab, -1000, 1000)

test_doubling_ratio(tri_comptage_mille, 10)
```

```
128 0.9444444445342711
256 1.2000000001940256
512 1.1764705881282946
1024 1.2380952378353824
2048 1.6785714286178313
```

Mesure de temps d'exécution de `tri_comptage` avec `test_doubling_ratio`:

```
def tri_comptage_mille(tab):
    """
    Pour mesurer le temps d'exécution avec test_doubling_ratio
    qui manipule des tableaux dont les valeurs sont entre -1000
    et 1000
    """
    return tri_comptage(tab, -1000, 1000)

test_doubling_ratio(tri_comptage_mille)
```

On doit obtenir des résultats similaires en ordre de grandeur à l'exemple ci-dessous (réalisé en dehors d'un notebook). On observe que le temps d'exécution double avec la taille des tableaux pris en entrée. On peut donc conjecturer que la **complexité** temporelle du *tri par comptage* est **linéaire** c'est-à-dire proportionnelle à la taille n du tableau en entrée.

Cette performance optimale (il faut moins examiner tous les éléments du tableau pour les trier) n'est possible que si on dispose d'une informations sur les valeurs possibles du tableau !

```
128 1.0743989961992313
256 0.870277917382644
512 1.2943577510571704
1024 1.6098955588341208
2048 1.4876719424536828
4096 1.5180886124726969
8192 2.4159512550154822
16384 1.8960229978174394
32768 1.9436002678090172
65536 2.161386612924146
131072 2.008289392838942
262144 2.0080475037850287
```