

Trier

INFORMATIQUE COMMUNE - TP n° 1.4 - Olivier Reynet

À la fin de ce chapitre, je sais :

- ✎ coder un algorithme de tri simple et explicite
- ✎ évaluer le temps d'exécution d'un algorithme avec la bibliothèque time
- ✎ générer un graphique légendé avec la bibliothèque matplotlib

On souhaite trier des listes Python, considérées ici comme des tableaux, avec des algorithmes différents (cf. algorithmes 1, 2, 3 et 4). Chaque algorithme de tri est implémenté par une fonction Python. Le prototype de ces fonctions est `my_sort(t)`, où `t` est un paramètre formel qui représente le tableau à trier.

```
def my_sort(t):  
    # tri du tableau  
    # for i in range(len(t))  
    #     t[i] = ...
```

Si le tri s'effectue en place (cas du tri par insertion et sélection par exemple), on n'a pas besoin de construire une autre liste pour le résultat trié : les modifications se font sur la liste passée en paramètres et on ne renvoie rien. Si le tri n'est pas en place, comme dans le cas du tri par comptage, il faut alors retourner le tableau construit par la fonction et qui contient résultat du tri.

A Trier et compter les opérations élémentaires

- A1. Coder les algorithmes de tri par sélection et par insertion en respectant le prototype défini à la question précédente.

Solution :

```
def selection_sort(t):  
    for i in range(len(t)):  
        min_index = i  
        for j in range(i + 1, len(t)):  
            if t[j] < t[min_index]:  
                min_index = j  
        swap(t, i, min_index)  
  
def insertion_sort(t):  
    for i in range(1, len(t)):  
        to_insert = t[i]  
        j = i
```

```
while t[j - 1] > to_insert and j > 0:
    t[j] = t[j - 1]
    j -= 1
t[j] = to_insert
```

- A2. Tester ces algorithmes sur une **même** liste Python de longueur 20 et contenant de types `int` choisis aléatoirement entre 0 et 100.

Solution :

```
N = 20
M = 100

t = [randrange(0, M+1) for _ in range(N)]
copy_t = t[:] # création d'une copie du tableau t
               # pas comme copy_t = t, référence seulement
selection_sort(copy_t)
print(copy_t)
copy_t = t[:]
insertion_sort(copy_t)
print(copy_t)
copy_t = t[:]
counting_sort(copy_t)
print(copy_t)
```

- A3. Peut-t-on trier des listes de chaînes de caractères avec ces mêmes codes? Tester cette possibilité à l'aide de la liste `["Zorglub", "Spirou", "Fantasio", "Marsupilami", "Marsu", "Samovar", "Zantafio"]`. Analyser les résultats. Pourquoi est-ce possible? Pourquoi n'est-ce pas possible?

Solution : Les tris par sélection ou par insertion sont des tris comparatifs (ou tris génériques) : ils utilisent la comparaison de deux éléments du tableau pour trier les éléments. Lorsque les éléments sont des entiers, cela fonctionne grâce à l'ordre sur les entiers naturels. Lorsque les éléments sont des chaînes de caractères, c'est l'ordre lexicographique qui est utilisé. Cet ordre s'appuie sur l'ordre alphabétique et la position du caractère dans la chaîne pour comparer deux chaînes. Python utilise implicitement l'ordre lexicographique lorsque on compare deux chaînes de caractères avec l'opérateur `<`. C'est pourquoi les tris par insertion, sélection ou bulles fonctionnent aussi dans ce cas.

La bibliothèque `matplotlib` permet de générer des graphiques à partir de données de type `list` qui constituent les listes des abscisses et des ordonnées associées. La démarche à suivre est de :

- importer la bibliothèque `from matplotlib import pyplot as plt`
- créer une figure `plt.figure()`
- tracer une courbe `plt.plot(x,y)` si `x` et `y` sont les listes des abscisses et des ordonnées associées. La bibliothèque trace les points `(x[i], y[i])` sur le graphique.
- ajouter les éléments de légende et de titre,
- montrer la figure ainsi réalisée `plt.show()`.

Algorithme 1 Tri par sélection

```

1: Fonction TRIER_SELECTION(t)
2:    $n \leftarrow \text{taille}(t)$ 
3:   pour  $i$  de 0 à  $n - 1$  répéter
4:      $\text{min\_index} \leftarrow i$                                 ▷ indice du prochain plus petit
5:     pour  $j$  de  $i + 1$  à  $n - 1$  répéter                    ▷ pour tous les éléments non triés
6:       si  $t[j] < t[\text{min\_index}]$  alors
7:          $\text{min\_index} \leftarrow j$                             ▷ c'est l'indice du plus petit non trié!
8:      $\text{échanger}(t[i], t[\text{min\_index}])$                     ▷ c'est le plus grand des triés!

```

Algorithme 2 Tri par insertion

```

1: Fonction TRIER_INSERTION(t)
2:    $n \leftarrow \text{taille}(t)$ 
3:   pour  $i$  de 1 à  $n-1$  répéter
4:      $\text{à\_insérer} \leftarrow t[i]$ 
5:      $j \leftarrow i$ 
6:     tant que  $t[j-1] > \text{à\_insérer}$  et  $j > 0$  répéter
7:        $t[j] \leftarrow t[j-1]$                                 ▷ faire monter les éléments
8:        $j \leftarrow j-1$ 
9:      $t[j] \leftarrow \text{à\_insérer}$                             ▷ insertion de l'élément

```

La bibliothèque `time` permet notamment de mesurer le temps d'exécution d'un code. Un exemple de code utilisant ces deux bibliothèques est donné ci-dessous. Le graphique qui en résulte est montré sur la figure 1.

Code 1 – Exemple d'utilisation des bibliothèques `time` et `matplotlib`

```

import time
from matplotlib import pyplot as plt

def to_measure(d):
    time.sleep(d) # Do nothing, wait for d seconds

# Simple use
tic = time.perf_counter()
to_measure(0.1)
toc = time.perf_counter()

print(f"Execution time : {toc - tic} seconds")

# Plotting results
timing = []
delay = [d / 1000 for d in range(1, 100, 5)]
for d in delay:
    tic = time.perf_counter()
    to_measure(d)
    toc = time.perf_counter()
    timing.append(toc - tic)

```

```
plt.figure()
plt.plot(delay, timing, color='cyan', label='fonction to_measure')
plt.xlabel('Delay', fontsize=18)
plt.ylabel("Execution time", fontsize=16)
plt.legend()
plt.show()
```

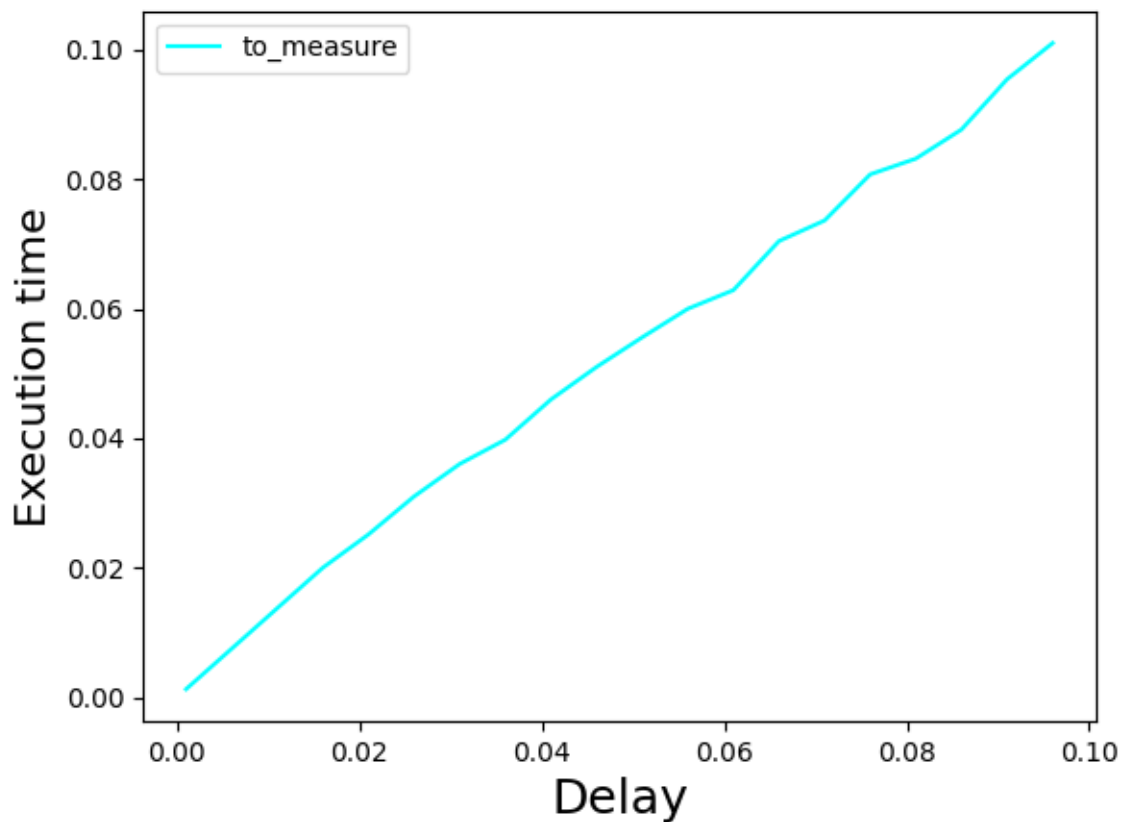


FIGURE 1 – Figure obtenue à partir des bibliothèques matplotlib et time et du code 1

- A4. À l'aide de la bibliothèque matplotlib, tracer les temps d'exécution nécessaires au tri d'un même tableau d'entiers par les algorithmes implémentés. On pourra également les comparer à la fonction `sorted` de Python. Analyser les résultats. Essayer de qualifier les coûts des algorithmes en fonction de la taille du tableau d'entrée.

Solution :**Code 2 – Trier des tableaux**

```
import time
from random import randint

def swap(t, i, j):
    t[i], t[j] = t[j], t[i]

def selection_sort(t):
    for i in range(len(t)):
        min_index = i
        for j in range(i + 1, len(t)):
            if t[j] < t[min_index]:
                min_index = j
        swap(t, i, min_index)

def insertion_sort(t):
    for i in range(1, len(t)):
        to_insert = t[i]
        j = i
        while t[j - 1] > to_insert and j > 0:
            t[j] = t[j - 1]
            j -= 1
        t[j] = to_insert

def bubble_sort(t):
    sorted = False
    i = len(t) - 1
    while i > 0 or not sorted:
        sorted = True
        for j in range(i):
            if t[j] > t[j + 1]:
                swap(t, j, j + 1)
                sorted = False
        i -= 1

def counting_sort(t):
    v_max = max(t)
    count = [0] * (v_max + 1)
    for e in t:
        count[e] += 1
    output = [None for i in range(len(t))]
    i = 0 # indice de parcours du tableau résultat
    for v in range(v_max + 1):
        for j in range(count[v]):
            output[i] = v
            i += 1
    return output
```

```

def complexity_counting_sort(t):
    v_max = max(t)
    count = [0] * (v_max + 1)
    for i in range(len(t)):
        count[t[i]] += 1
    for i in range(1, v_max + 1):
        count[i] += count[i - 1]
    output = [None for i in range(len(t))]
    for i in range(len(t)):
        output[count[t[i]] - 1] = t[i]
        count[t[i]] -= 1
    return output

def counting_sort_bis(t):
    if len(t) > 0:
        v_max = max(t)
        # v_max = t[0]
        # for i in range(1, len(t)):
        #     if t[i] > v_max:
        #         v_max = t[i] # trouver le maximum
        count = [0 for _ in range(v_max + 1)]
        for i in range(len(t)):
            count[t[i]] += 1 # compter les occurrences de chaque t[i] de t.
            # Certaines cases de count sont nulles : pas de valeur
            # correspondante dans t
        for i in range(1, v_max + 1):
            count[i] += count[i - 1] # cumul des effectifs
        output = [None for _ in range(len(t))]
        for i in range(len(t)):
            count[t[i]] -= 1 # on compte à partir de 0
            output[count[t[i]]] = t[i] # à la bonne place !
            # print("elem :", t[i], "place :", count[t[i]], output)
        return output
    else:
        return None

def sort_timing():
    sizes = [i for i in range(10, 5000, 500)]
    M = 10 * max(sizes)
    results = []
    for i in range(len(sizes)):
        t = [randint(0, M) for _ in range(sizes[i])]
        mem_t = t[:]
        results.append([])
        for method in [selection_sort, bubble_sort, insertion_sort,
                        counting_sort, sorted]:
            t = mem_t[:]
            tic = time.perf_counter()
            method(t)
            toc = time.perf_counter()
            results[i].append(toc - tic)
        # print("#", i, " - ", sizes[i], " -> ", results)

```

```

sel = [results[i][0] for i in range(len(sizes))]
bub = [results[i][1] for i in range(len(sizes))]
ins = [results[i][2] for i in range(len(sizes))]
cnt = [results[i][3] for i in range(len(sizes))]
p = [results[i][4] for i in range(len(sizes))]

from matplotlib import pyplot as plt

plt.figure()
plt.plot(sizes, sel, color='red', label='selection sort')
plt.plot(sizes, bub, color='green', label='bubble sort')
plt.plot(sizes, ins, color='blue', label='insertion sort')
plt.plot(sizes, cnt, color='cyan', label='counting sort')
plt.plot(sizes, p, color='black', label='sorted python')

plt.xlabel('n', fontsize=18)
plt.ylabel('time', fontsize=16)
plt.legend()
plt.show()

# MAIN PROGRAM
N = 20
M = 100

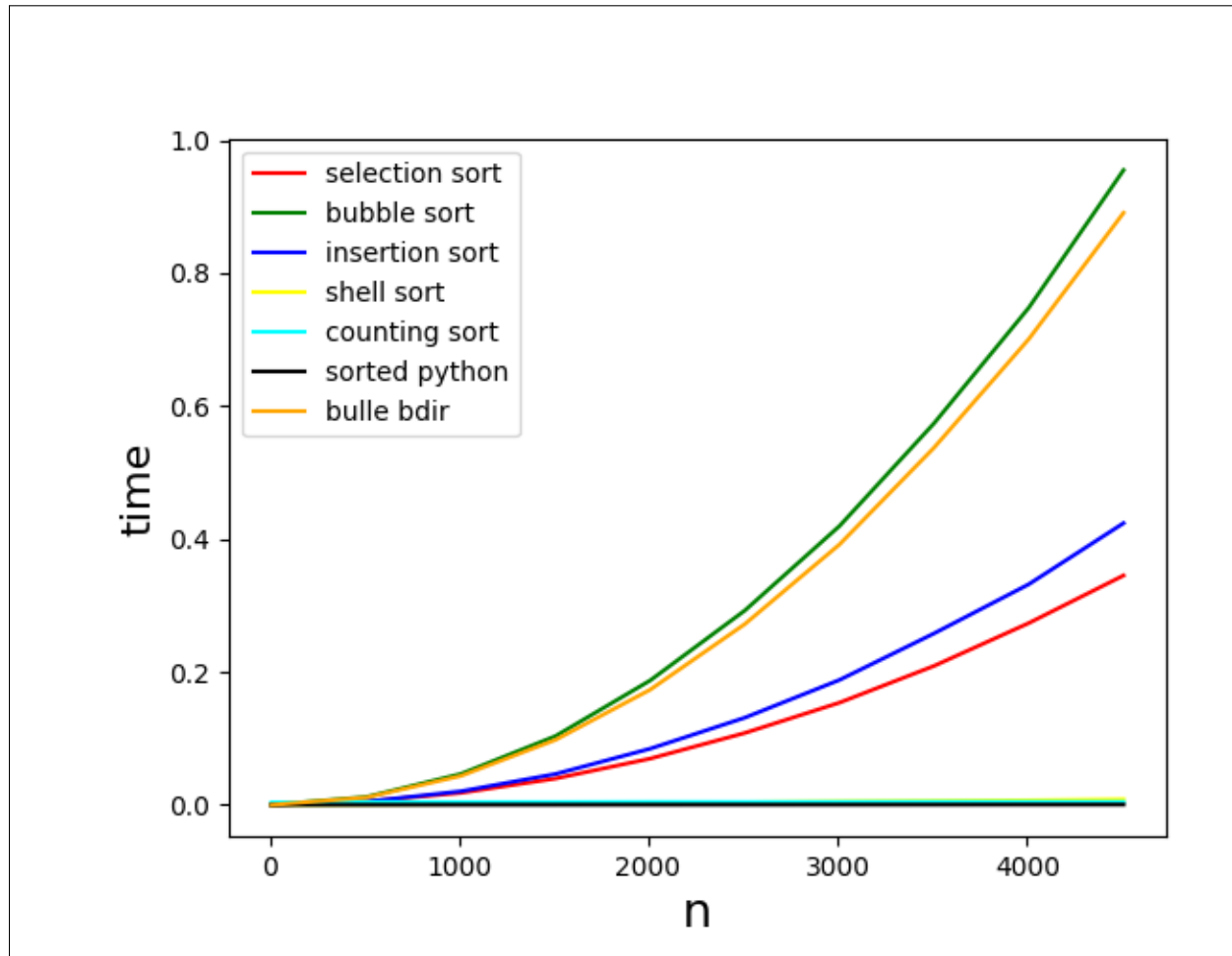
t = [randint(0, M) for _ in range(N)]

for method in [selection_sort, insertion_sort, bubble_sort, counting_sort]:
    tmp = t[:]
    if method in [selection_sort, bubble_sort, insertion_sort]: # in-place
        method(tmp)
    else:
        tmp = method(tmp) # counting_sort not in-place
    print("--> Method", method.__name__, ":\n ", t, "\n", tmp)

t = ["Zorglub", "Spirou", "Fantasio", "Marsupilami", "Marsu", "Samovar", "
Zantafio"]
for method in [selection_sort, insertion_sort, bubble_sort]: # , counting_sort
]:
    method(t)
    print("--> Method", method.__name__, ":\n ", t, "\n", t)

sort_timing()

```



B Tri par comptage

Le tri par comptage s'applique à des valeurs entières dont l'algorithme fait tout d'abord l'histogramme. Dans un second temps, il construit un tableau trié dont les valeurs sont celles du tableau d'entrée en s'appuyant sur l'histogramme.

B1. Écrire une fonction de signature `counting_sort(t : list[int])` qui implémente le tri par comptage.

Solution :

```
def counting_sort(t):
    v_max = max(t)
    count = [0] * (v_max + 1)
    for e in t:
        count[e] += 1
    output = [None for i in range(len(t))]
    i = 0 # indice de parcours du tableau résultat
    for v in range(v_max + 1):
        for j in range(count[v]):
```


Algorithme 3 Tri par comptage

```

1: Fonction TRIER_COMPTAGE( $t, v_{max}$ )                                ▷  $v_{max}$  est le plus grand entier à trier
2:    $n \leftarrow \text{taille}(t)$ 
3:    $c \leftarrow$  un tableau de taille  $v_{max} + 1$  initialisé avec des zéros
4:   pour  $i$  de 0 à  $n - 1$  répéter
5:      $c[t[i]] \leftarrow c[t[i]] + 1$                                 ▷ compter les occurrences de chaque élément du tableau.
6:    $\text{résultat} \leftarrow$  un tableau de taille  $n$ 
7:    $i \leftarrow 0$ 
8:   pour  $v$  de 0 à  $v_{max}$  répéter                                ▷ On prend chaque valeur possible dans l'ordre
9:     si  $c[v] > 0$  alors                                          ▷ Si l'élément  $v$  est présent dans le tableau
10:      pour  $j$  de 0 à  $c[v] - 1$  répéter
11:         $\text{résultat}[i] \leftarrow v$                                 ▷ alors écrire autant de  $v$  que d'occurrences de  $v$ 
12:         $i \leftarrow i + 1$                                           ▷ à la bonne place, la ième!
13:   renvoyer  $\text{résultat}$ 

```

```

        output[i] = v
        i += 1
    return output

```

B2. Comparer le temps d'exécution de ce tri aux tris de la section précédente.

B3. Quels sont les avantages et les inconvénients de ce tri?

Solution :

Le tri par comptage n'est pas un tri comparatif (ou générique) : c'est un tri par dénombrement de valeurs entières. Il ne porte donc que sur des tableaux contenant des entiers et ne peut pas fonctionner pour des chaînes de caractères par exemple.

Il ne s'applique qu'à des entiers, mais sa complexité est en $O(n)$.

C Tri shell --> HORS PROGRAMME

Lors d'un tri par insertion, de nombreuses opérations sont nécessaires pour insérer un élément dans la partie triée. Par contre, si la liste est quasiment triée, peu d'opérations sont nécessaires.

Le tri shell permet d'améliorer les performances du tri par insertion. Dans ce tri, chaque liste d'éléments séparés de N positions est triée avec le tri par insertion. L'algorithme effectue plusieurs fois cette opération en diminuant la valeur de N jusqu'à $N = 1$, ce qui équivaut alors à trier tous les éléments du tableau.

C1. Écrire une fonction de signature `shell_sort(t : list[int])` qui implémente le tri shell.

C2. Comparer les performances de ce tri à celles des algorithmes de la première section.

D Des bulles et des cocktails --> HORS PROGRAMME

Le tri bulle est un tri par comparaisons et échanges, stable et en place. Il n'est pas efficace sauf dans le cas où le tableau à trier est quasiment trié. Sa vocation reste essentiellement pédagogique.

Le principal avantage du tri bulle est qu'il est simple à expliquer. Son principe est le suivant : on prend chaque élément du tableau et on le fait monter d'une place dans le tableau si celui-ci est plus grand que son successeur en échangeant leurs places. Sinon, on fait prendre le successeur et on opère à l'identique.

Algorithme 4 Tri bulle

```

1: Fonction PUT_VMAX_IN_LAST_PLACE(t, last)
2:   pour j de 0 à last-1 répéter
3:     si t[j] > t[j+1] alors
4:       SWAP(t[j], t[j+1])                                ▷ faire monter la bulle
5: Fonction BUBBLE_SORT(t)
6:   n ← taille(t)
7:   pour i de n-1 à 1 répéter
8:     PUT_VMAX_IN_LAST_PLACE(t, i)

```

D1. Écrire une fonction de signature `put_vmax_in_last_place(t, last)` qui place l'élément le plus grand de la partie du tableau à trier à sa place (indice `last`).

Solution :

```

def put_vmax_in_last_place(t, last):
    for j in range(last):
        if t[j] > t[j + 1]:
            swap(t, j, j + 1)

```

D2. Écrire une fonction de signature `bubble_sort(t : list[int])` qui implémente le tri bulle.

Solution :

```

def bubble_sort(t):
    for i in range(len(t) - 1, -1, -1):
        put_vmax_in_last_place(t, i)

```

D3. Comparer les performances de ce tri à celles des algorithmes de la première section.

D4. Améliorer cette algorithme afin de rendre le tri bulle bidirectionnel : faire descendre un élément une fois qu'on en a monté un. Cette variante s'appelle le tri cocktail.

Solution :

```

def put_vmin_in_first_place(t, last):
    for j in range(last, 0, -1):
        if t[j] < t[j - 1]:
            swap(t, j, j - 1)

```

```
def cocktail_sort(t):  
    for i in range(len(t) - 1, 0, -2):  
        put_vmin_in_first_place(t, i)  
        put_vmax_in_last_place(t, i)
```

D5. Implémenter le tri à peigne qui reprend l'idée du tri shell en l'appliquant au tri bulle : la comparaison ne se fait pas avec le voisin mais avec un autre élément éloigné de N places. On recommence en réduisant la valeur de N jusqu'à ce que $N = 1$.

Solution :

```
def shell_sort(t):  
    n = len(t)  
    gap = n // 2  
    while gap > 0:  
        for i in range(gap, n):  
            to_insert = t[i]  
            j = i  
            while j >= gap and t[j - gap] > to_insert:  
                t[j] = t[j - gap]  
                j -= gap  
            t[j] = to_insert  
        gap = gap // 2
```
