Trier

INFORMATIQUE COMMUNE - TP n° 1.4 - Olivier Reynet

À la fin de ce chapitre, je sais :

- ☞ coder un algorithme de tri simple et explicité
- ☞ 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] = ...
```

Ⓡ Il est important de distinguer les tris en place des autres tris.

- 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 Python passée en paramètre et la fonction de tri ne renvoie rien.**
- Si le tri n'est pas en place, comme dans le cas du tri par comptage, il faut alors **retourner** la liste Python construite par la fonction et qui contient résultat du tri.

A Tris génériques et nombre d'opérations élémentaires

A1. Coder les algorithmes de tri par sélection (cf. algorithme 1) et par insertion (cf. algorithme 2) 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)
```

```python
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 :

```python
N = 20
M = 100

t = [radnrange(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 (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.

D'une manière générale, un tri générique (ou comparatif) est possible dès lors qu'on l'on sait comparer deux éléments de l'ensemble que l'on souhaite trier.

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 :

Algorithme 1 Tri par sélection

```
1 : Fonction TRIER_SELECTION(t)
2 :     n ← taille(t)
3 :     pour i de 0 à n − 1 répéter
4 :         min_index ← 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[min_index ] alors
7 :                 min_index ← j                  ▷ c'est l'indice du plus petit non trié!
8 :         échanger(t[i], t[min_index])           ▷ c'est le plus grand des triés!
```

Algorithme 2 Tri par insertion

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

- 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()`.

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.

```
import time
from random import randrange
from matplotlib import pyplot as plt

tailles = [i for i in range(10, 5000, 500)]
temps = []
for i in range(len(tailles)):
    t = [randrange(0, M + 1) for _ in range(tailles[i])]
    temps.append([])
    tic = time.perf_counter()
    insertion_sort(t)
    toc = time.perf_counter()
    temps[i].append(toc - tic)

plt.figure()
plt.plot(tailles, temps, color='red', label='insertion sort')
plt.xlabel('size')
```

```
plt.ylabel('time')
plt.legend()
plt.show()
```

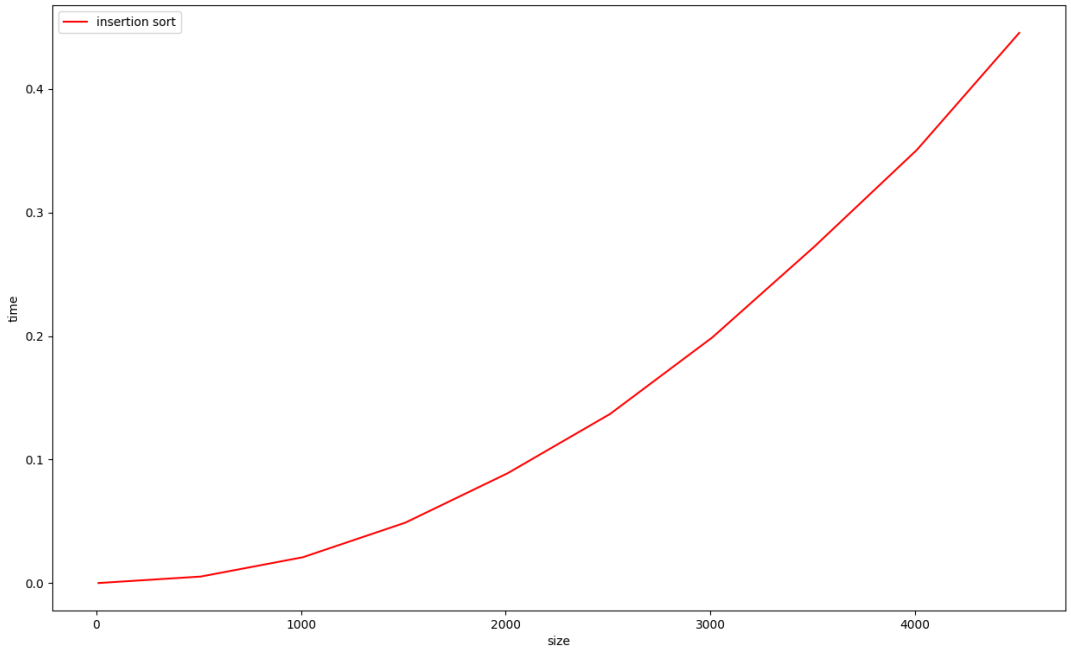


FIGURE 1 – Figure obtenue à partir des bibliothèques matplotlib et time et du code d'exemple

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. La taille du tableau considéré varie de 10 à 5000 exclu par pas de 500. 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 :

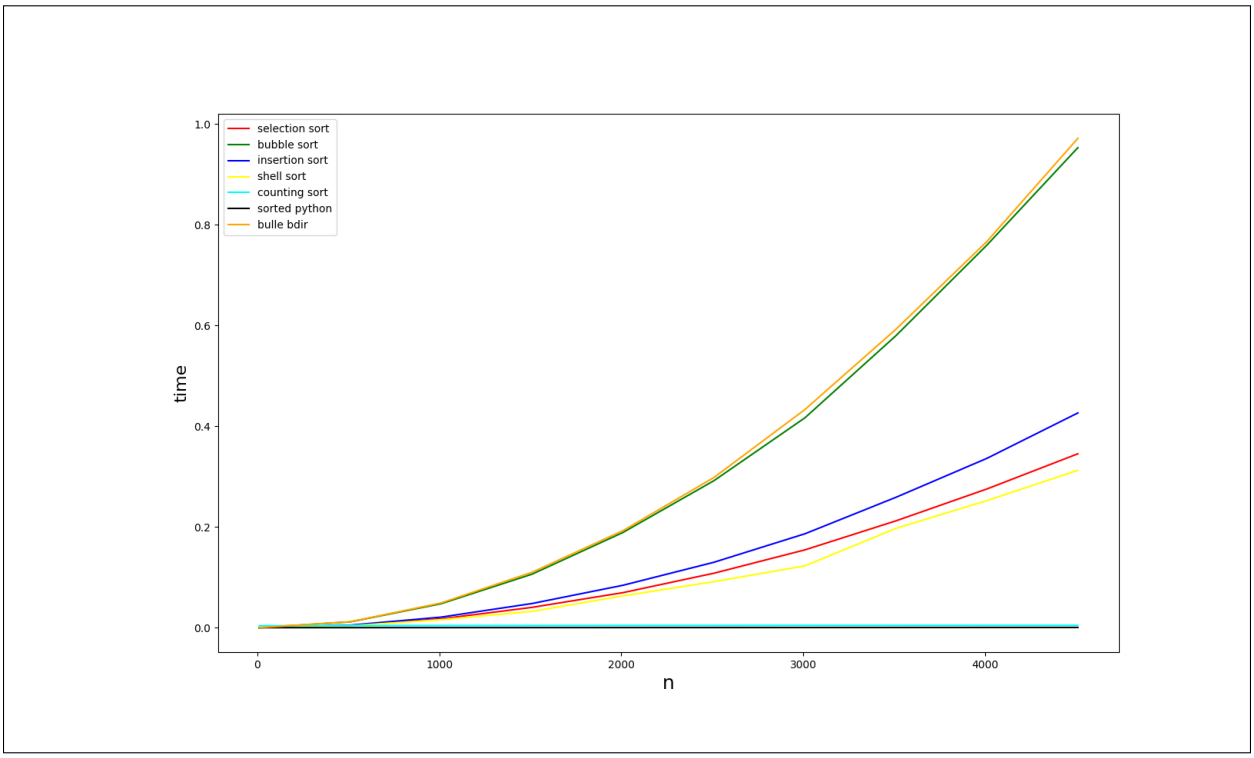
```
from matplotlib import pyplot as plt

def simple_timing():
    sizes = [i for i in range(10, 5000, 500)]
    M = 10 * max(sizes)
    results = []
    for i in range(len(sizes)):
        t = [randrange(0, M + 1) for _ in range(sizes[i])]
        mem_t = t[:]
        results.append([])
        for method in [selection_sort, insertion_sort, sorted]:
            t = mem_t[:]
            tic = time.perf_counter()
            method(t)
            toc = time.perf_counter()
            results[i].append(toc - tic)

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

    plt.figure()
    plt.plot(sizes, sel, color='red', label='selection sort')
    plt.plot(sizes, ins, color='blue', label='insertion sort')
    plt.plot(sizes, p, color='black', label='sorted python')

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



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.

Algorithme 3 Tri par comptage

```
1 : Fonction TRIER_COMPTAGE(t, v_max)                ▷ v_max est le plus grand entier à trier
2 :     n ← taille(t)
3 :     c ← un tableau de taille v_max + 1 initialisé avec des zéros
4 :     pour i de 0 à n − 1 répéter
5 :         c[t[i]] ← c[t[i]] + 1                    ▷ compter les occurrences de chaque élément du tableau.
6 :     résultat ← un tableau de taille n
7 :     i ← 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 1 à c[v] répéter           ▷ Répéter autant de fois que d'occurrences
11 :                résultat[i] ← v                  ▷ alors écrire autant de v que d'occurrences de v
12 :                i ← i + 1                        ▷ à la bonne place, la ième!
13 :    renvoyer résultat
```

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:  # création de l'histogramme
        count[e] += 1
    output = []
    for v in range(v_max + 1):
        for j in range(count[v]):  # Exploitation de l'histogramme
            output.append(v)
    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 le pire des cas, c'est-à-dire lorsque la liste n'est pas triée. Dans ce tri, chaque liste d'éléments séparés de N positions est triée avec le tri par insertion. On commence généralement avec $N = \lfloor n/2 \rfloor$ puis on divise par deux N à chaque étape. L'algorithme effectue donc plusieurs fois cette opération en diminuant jusqu'à ce que $N = 1$, 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.

Solution :

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

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

Solution : Le tri shell présente de meilleures performances que le tri par insertion lorsque la liste n'est pas triée. Le choix de l'espacement entre les éléments triés (la suite de N) est crucial pour garantir une bonne complexité.

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 prend 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 l'élément le plus petit élément une fois qu'on en a monté le plus grand. Cette variante s'appelle le tri cocktail.

Solution :

```
def bidir_bubble_sort(t):  # cocktail !
    start = 0
    last = len(t) - 1
    while start < last:
        for j in range(start, last):
            if t[j] > t[j + 1]:
                swap(t, j, j + 1)
        last -= 1
        for j in range(last, start, -1):
            if t[j] < t[j - 1]:
                swap(t, j, j - 1)
        start += 1
```