Trier et rechercher

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
- ☞ rechercher un élément dans un tableau séquentiellement ou par dichotomie itérative
- ☞ générer un graphique légendé avec la bibliothèque matplotlib

A Trier un tableau

A1. On souhaite trier des listes Python, considérées ici comme des tableaux, avec des algorithmes différents (cf. algorithmes 1, 2 et 3). 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] = ...
```

Cette fonction, une fois réalisée, trie le tableau `t` passé en paramètre mais ne renvoie rien (i.e. pas de `return`). Expliquer pourquoi.

A2. Coder les algorithmes de tri par sélection, par insertion et par comptage en respectant le prototype défini à la question précédente [1].

A3. Tester ces algorithmes sur une **même** liste Python de longueur 20 et contenant de types `int` choisis aléatoirement entre 0 et 100.

A4. Peut-t-on trier des listes de chaînes de caractères avec ces mêmes codes ? Tester cette possbilité 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 ?

La bibliothèque `matplotlib` permet de générer des graphiques à partir de données de type `list` qui constituent les abscisses et les 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.

1. On a le droit de collaborer, de se répartir les algorithmes et de s'échanger les codes s'ils sont corrects !

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
```

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 0 à c[v] - 1 répéter
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
```

- 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.

Code 1 – Example 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()
```

A5. 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.

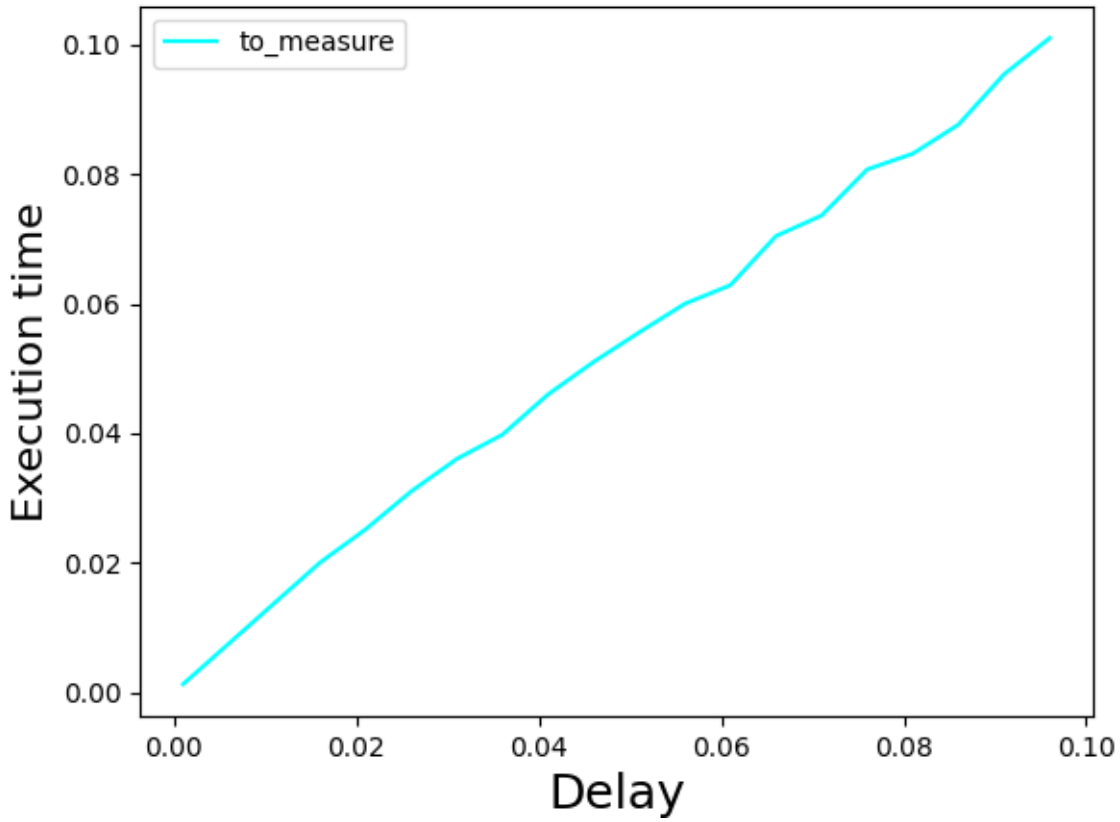

FIGURE 1 – Figure obtenue à partir des bibliothèques matplotlib et time et du code 1

B Recherche d'un élément dans un tableau

On considère une liste `L` contenant des éléments de type `int`. Cette liste est **triée** par ordre croissant de ses éléments. On veut savoir si un élément `x` est présent dans `L` et comparer les performances des approches séquentielles et la dichotomiques. On dispose des algorithmes 4, 5 et 6.

Algorithme 4 Recherche séquentielle d'un élément dans un tableau

```
1 : Fonction RECHERCHE_SÉQUENTIELLE(t, elem)
2 :     n ← taille(t)
3 :     pour i de 0 à n − 1 répéter
4 :         si t[i]= elem alors
5 :             renvoyer i                    ▷ élément trouvé, on renvoie sa position dans t
6 :     renvoyer l'élément n'a pas été trouvé
```

Algorithme 5 Recherche d'un élément par dichotomie dans un tableau trié

```
1 : Fonction RECHERCHE_DICHOTOMIQUE(t, elem)
2 :     n ← taille(t)
3 :     g ← 0
4 :     d ← n-1
5 :     tant que g ⩽ d répéter                ▷ ⩽ cas où valeur au début, au milieu ou à la fin
6 :         m ← (g+d)//2                      ▷ Division entière : un indice est un entier!
7 :         si t[m] < elem alors
8 :             g ← m + 1                     ▷ l'élément devrait se trouver dans t[m+1, d]
9 :         sinon si t[m] > elem alors
10 :            d ← m - 1                     ▷ l'élément devrait se trouver dans t[g, m-1]
11 :        sinon
12 :            renvoyer m                    ▷ l'élément a été trouvé
13 :    renvoyer l'élément n'a pas été trouvé
```

B1. Coder l'algorithme de recherche séquentielle d'un élément dans un tableau. Lorsque l'élément n'est pas présent dans le tableau, la fonction Python renvoie `None`. Sinon, elle renvoie l'indice de l'élément trouvé dans le tableau. Vérifier que cet algorithme fonctionne sur un tableau d'entiers de 20 éléments rempli aléatoirement. Dans le pire des cas, quel est le coût d'une recherche séquentielle en fonction de la taille du tableau ?

B2. Coder l'algorithme 5. Lorsque l'élément n'est pas présent dans le tableau, la fonction Python renvoie `None`. Sinon, elle renvoie l'indice de l'élément trouvé dans le tableau. Vérifier que cet algorithme fonctionne sur un tableau d'entiers de 20 éléments rempli aléatoirement et trié.

B3. Coder l'algorithme 6. Vérifier que cet algorithme fonctionne sur un tableau d'entiers de 20 éléments rempli aléatoirement et trié et que l'indice renvoyé est bien l'indice minimal de la première occurrence de l'élément recherché.

B4. On suppose que la longueur de la liste est une puissance de 2, c'est à dire $n = 2^p$ avec $p \geqslant 1$. Combien d'étapes l'algorithme 6 comporte-t-il? En déduire le nombre de comparaisons effectuées, dans le

Algorithme 6 Recherche d'un élément par dichotomie dans un tableau trié, renvoyer l'indice minimal en cas d'occurrences multiples.

```
1 : Fonction RECHERCHE_DICHOTOMIQUE_INDICE_MIN(t, elem)
2 :     n ← taille(t)
3 :     g ← 0
4 :     d ← n-1
5 :     tant que g < d répéter                    ▷ attention au strictement inférieur!
6 :         m ← (g+d)//2                          ▷ Un indice de tableau est un entier!
7 :         si t[m] < elem alors
8 :             g ← m + 1                         ▷ l'élément devrait se trouver dans t[m+1, d]
9 :         sinon
10 :            d ← m                             ▷ l'élément devrait se trouver dans t[g, m]
11 :    si t[g] = elem alors
12 :        renvoyer g
13 :    sinon
14 :        renvoyer l'élément n'a pas été trouvé
```

cas où l'élément est absent, en fonction de p puis de n, et comparer avec l'algorithme de recherche séquentielle.

B5. Tracer le graphique des temps d'exécution des algorithmes précédent en fonction de n. Les tracés sont-ils cohérents avec les calculs des coûts effectués précédemment ?

B6. La recherche dichotomique fonctionne-t-elle sur les listes non triées ? Donner un contre-exemple si ce n'est pas le cas.