
TRAVAUX PRATIQUES n° 11

Algorithmes de tris

Dans ce TP, on apprend à programmer des algorithmes de tri élémentaires (sélection, insertion, comptage), puis on voit l'intérêt d'avoir une liste triée : la recherche par dichotomie. On termine par une application à l'ordre alphabétique et, pour aller plus loin, par le tri rapide.

Exercice 1 (*Tri par sélection*) :

- 1) Écrire une fonction `tri_par_selection(L)` programmant l'algorithme du tri par sélection pour trier une liste de nombres dans l'ordre croissant en s'aidant du pseudo-code suivant :

```
"""
def tri_par_selection(L) :
    on note n = longueur de L
    pour tout i allant de 0 à n-1 (bornes incluses) :
        - trouver la position j de l'élément le plus petit de L[i:n] dans L
        - échanger L[i] et L[j]
    retourner L
"""
```

- 2) Écrire une fonction `tri_par_selection_decroissant(L)` adaptant l'algorithme précédent pour trier une liste de nombres dans l'ordre décroissant.

Exercice 2 (*Tri par insertion*) :

- 1) Écrire une fonction `insertion(L, i)` qui prend en arguments une liste `L` et un indice `i` tel que $0 \leq i < \text{len}(L)$. On suppose que la sous-liste `L[0:i]` est déjà triée dans l'ordre croissant. La fonction insère l'élément `L[i]` à sa place parmi les `i` premiers éléments. Cette fonction ne doit rien renvoyer mais doit modifier la liste `L`.

Par exemple, si `L = [1, 2, 4, 6, 3, 5]`, alors

```
>>> insertion(L, 4)
>>> print(L)
[1, 2, 3, 4, 6, 5]
>>> insertion(L, 5)
>>> print(L)
[1, 2, 3, 4, 5, 6]
```

- 2) Écrire une fonction `tri_par_insertion(L)` programmant l'algorithme de tri par insertion en utilisant la fonction `insertion` précédente.

Exercice 3 (*Tri par comptage*) : On suppose dans cet exercice que les éléments de `L` sont des entiers appartenant à $\llbracket 0, n \rrbracket$.

- 1) Écrire une fonction `effectifs(L, n)` qui prend en arguments une liste `L` et un entier $n \geq 0$ et qui renvoie une liste `E` de longueur $n+1$ telle que, pour tout $k \in \llbracket 0, n \rrbracket$, `E[k]` est le nombre d'occurrences de `k` dans `L`.

Par exemple, si `L = [3, 1, 2, 5, 3, 1, 5, 4, 4, 4, 5]`, alors

```
>>> effectifs(L, 6)
[0, 2, 1, 2, 3, 3, 0]
```

- 2) Écrire une fonction `tri_par_comptage(L, n)` programmant l'algorithme de tri par comptage en utilisant la fonction `effectifs` précédente. Cette fonction renvoie une nouvelle liste triée dans l'ordre croissant.

Exercice 4 (*Recherche par dichotomie*) : Une fois une liste triée, on peut rechercher un élément beaucoup plus rapidement qu'en parcourant toute la liste. C'est l'un des intérêts majeurs des algorithmes de tri.

- 1) Écrire une fonction `recherche(L, e)` qui prend en argument une liste `L` et un nombre `e` et qui renvoie `True` si `e` est un élément de `L` et `False` sinon (sans utiliser la syntaxe `if e in L`).
- 2) Lorsqu'une liste est triée dans l'ordre croissant, on peut rendre la fonction précédente beaucoup plus efficace en faisant une recherche par dichotomie. Écrire une fonction `recherche_par_dichotomie(L, e)` qui prend en argument une liste `L` **déjà triée** dans l'ordre croissant et un nombre `e`, et qui renvoie `True` si `e` est un élément de `L` et `False` sinon. On utilisera le pseudo-code suivant :

```
"""
def recherche_par_dichotomie(L, e) :
    debut = 0
    fin = len(L)-1
    tant que debut <= fin :
        milieu = (debut+fin)//2
        si L[milieu] est égal à e
            retourner vrai
        sinon si L[milieu] < e
            debut = milieu+1
        sinon
            fin = milieu-1
    retourner faux
"""
```

- 3) Pour comparer les deux algorithmes, on pourra taper dans le shell les commandes suivantes (*attention : la création d'une liste de 10^8 éléments demande beaucoup de mémoire ; on pourra commencer avec 10^6 ou 10^7 si besoin*) :

```
>>> import numpy as np
>>> L = np.random.randint(0, 10**9, 10**7) # Liste aléatoire d'entiers
>>> recherche(L, 1) # Peut prendre quelques secondes
>>> L.sort() # Peut aussi prendre quelques secondes
>>> recherche_par_dichotomie(L, 1) # Devrait être quasi instantanée
```

Exercice 5 (*Tri par ordre alphabétique*) :

- 1) Écrire une fonction `plus_petit(chaine1, chaine2)` prenant en arguments deux chaînes de caractères (en minuscules, sans accents) et qui renvoie `True` si `chaine1` est avant `chaine2` pour l'ordre alphabétique, et `False` sinon.

On n'utilisera pas la comparaison native `chaine1 < chaine2`. On pourra utiliser l'instruction `alphabet = [chr(k) for k in range(97, 123)]` pour créer une liste contenant toutes les lettres de l'alphabet dans l'ordre.

- 2) Écrire une fonction `tri_mot(L)` prenant en argument une liste de chaînes de caractères et qui renvoie la liste triée dans l'ordre alphabétique.

On utilisera le tri de son choix, en s'aidant de `plus_petit`.

- 3) Écrire une fonction `plus_grand_anagramme(mot)` prenant en argument une chaîne de caractères `mot` et qui renvoie le plus grand anagramme de `mot` pour l'ordre alphabétique.

Exercice 6 (*Tri rapide (pour aller plus loin)*) : Écrire une fonction `tri_rapide(L)` programmant l'algorithme de tri rapide à l'aide du pseudo-code suivant :

```
"""
def tri_rapide(L) :
    si L est vide ou de longueur 1 :
        retourner L
    on enlève le premier élément de L et on le note pivot
    petits = []
    grands = []
    pour tout élément e de L :
        si e < pivot :
            on ajoute e à la liste petits
        sinon :
            on ajoute e à la liste grands
    on trie les listes petits et grands à l'aide de la fonction tri_rapide
    on reconstitue la liste triée (petits + [pivot] + grands) que l'on renvoie
"""
```