Chapitre 6

Rappel sur les tris

I. Tri à bulle

Il consiste à comparer répétitivement les éléments consécutifs d'un tableau, et à les permuter lorsqu'ils sont mal triés. Il doit son nom au fait qu'il déplace rapidement les plus grands éléments en fin de tableau, comme des bulles d'air qui remonteraient rapidement à la surface d'un liquide.
Le tri à bulles est souvent enseigné en tant qu'exemple algorithmique, car son principe est simple. Mais c'est le plus lent des algorithmes de tri communément enseignés, et il n'est donc guère utilisé en pratique.

Rappel des différentes étapes : :

- on parcourt la liste et si deux éléments consécutifs sont rangés dans le désordre, on les échange ;
- si à la fin du parcours au moins un échange à eu lieu, on recommence l'opération ;
- sinon, la liste est triée, on arrête.

Implémentation : :

```
def tri_bulle(L):
    for j in range(len(L)-1, 0, -1):
        echange = 0
        for i in range(1, j+1):
            if L[i-1] > L[i]:
                L[i-1], L[i] = L[i], L[i-1]
                echange = 1
        if echange == 0:
            break
```

II. Manipulations de liste

Exercice 6.1
Écrire une fonction `est_triee(L)` qui renvoie `True` si la liste `L` est triée par ordre croissant, et `False` sinon.

Exercice 6.2
Écrire une fonction `echange(L, i, j)` qui modifie L en échangeant les éléments aux positions i et j, et ne renvoie rien.
Par exemple :

```
>>> L = [2, 3, 7, 8, 4, 1]
>>> echange(L, 1, 3)
>>> L
```

```
[2, 8, 7, 3, 4, 1]
```

Exercice 6.3

Écrire une fonction `insere(L, i)` qui réalise l'insertion de la valeur `L[i]` dans la partie de la liste `L[0], L[1], ..., L[i-1]` en supposant que les valeurs `L[0], L[1], ..., L[i-1]` sont triées.
Par exemple :

```
>>> L = [2, 3, 7, 8, 5, 1]
>>> insere(L, 4, 5)
>>> L
[2, 3, 5, 7, 8, 1]
>>> insere(L, 5, 1)
[1, 2, 3, 5, 7, 8]
```

III. Caractéristiques des tris

- **Clé du tri** : c'est la relation d'ordre total sur un élément de la structure que l'on trie. Cela peut être la relation d'ordre sur les entiers ou réels mais cela peut également être l'ordre lexicographique sur les chaînes de caractères, ou les tuples.
- **Tri en place** : c'est un tri qui lit et écrit directement dans la structure triée. Elle utilise donc peu de mémoire supplémentaire.
- **Tri stable** : c'est un tri qui lorsqu'il rencontre des éléments ayant la même clé dans la structure initiale, les place dans le même ordre dans la structure triée.

IV. Tri par sélection

Le tri par sélection consiste à :

- Rechercher le plus petit élément de la liste et l'échanger avec la valeur située en première position ;
- Rechercher le deuxième plus petit élément de la liste et l'échanger avec la valeur située en deuxième position ;
- Répéter ce procédé jusqu'à ce que tous les éléments aient été placés.

Exercice 6.4

1. Appliquer cet algorithme pour trier la liste `L = [3, 4, 1, 7, 2]`
2. Écrire une fonction `indice_min(L, indice_debut)` qui retourne l'indice de la valeur minimale de la liste où `indice_debut` est l'indice à partir duquel on commence la recherche. Par exemple :

   ```
   >>> L = [2, 3, 7, 8, 5, 6]
   >>> indice_min(L, 2)
   4
   ```

3. En déduire une fonction triant une liste par sélection
4. Quelles sont les caractéristiques de ce tri
5. Evaluer la complexité de cette fonction

Implémentation du tri par sélection : :

```
def indice_du_min(L, indice_debut):
    '''détermine l'indice du minimum de la liste à partir de l'indice indice_debut'''
    ind_du_min = indice_debut
    for i in range(indice_debut + 1, len(L)):
        if L[i] < L[ind_du_min]:
            ind_du_min = i
    return ind_du_min

def tri_par_selection(L):
    ''' tri la liste par ordre croissant en utilisation la méthode par selection
    le tri est en place'''
    for i in range(len(L)-1):
        ind_du_min = indice_du_min(L, i)
        echange(L, i, ind_du_min)
```

V. Tri par insertion

Le tri par insertion est le tri souvent utilisé par les joueurs de carte. Il suit le même principe que le tri par sélection, en parcourant la liste de la gauche vers la droite et en maintenant une partie déjà triée sur la gauche.
Mais plutôt que de cherche la plus petite valeur dans la partie non encore triée, le tri par insertion va insérer la première valeur non encore triée (c'est à dire la plus à gauche dans la partie non triée) dans la partie déjà triée. Exemple avec L = [5, 3, 2, 7, 4] :

- 1ère étape : [**5**, 3, 2, 7, 4]. Cette étape consistant à insérer la valeur 5 sur la partie de gauche peut-être évitée
- 2ème étape : [**3**, **5**, 2, 7, 4]
- 3ème étape : [**2**, **3**, **5**, 7, 4]
- 4ème étape : [**2**, **3**, **5**, **7**, 4]
- 5ème étape : [**2**, **3**, **4**, **5**, **7**]

> **Exercice 6.5**
>
> 1. Appliquer cet algorithme à la liste `[3, 4, 1, 7, 2]`
> 2. Écrire une fonction triant une liste par cet algorithme.
> 3. Quelles sont les caractéristiques de ce tri
> 4. Évaluer sa complexité.

Implémentation du tri par insertion :

```
def insere(L, i):
    ''' insere L[i] dans L[0..i[
        on suppose L[0..i[ triée '''
    j = i
    while j > 0 and L[i] < L[j-1]:
        L[j] = L[j-1]
        j = j - 1
    L[j] = L[i]

def tri_insertion(L):
    ''' tri la liste par ordre croissant en utilisation la méthode par insertion
    le tri est en place'''
    for i in range(1, len(L)):
        insere(L, i)
```

VI. Tri fusion

Le principe « diviser pour régner » peut être avantageusement utilisé pour concevoir un algorithme de tri plus efficace encore appelé *tri fusion*. Le principe consiste à séparer la liste en deux listes de même taille à un élément près puis à trier chacune de ces liste avec le tri fusion récursivement. Enfin on fusionne les deux listes triées. Pour cela il suffit de d'examiner le premier élément de chaque liste, de la placer dans une nouvelle liste puis poursuivre cette opération jusqu'à épuisement de l'une des liste. On complète alors notre nouvelle liste en ajoutant à la fin de celle-ci les éléments de la liste non vide.
Il s'agit bien là du principe « diviser pour régner » car on ramène le problème du tri d'une liste aux sous-problème du tri de deux listes plus petites jusqu'à parvenir à des listes d'au plus un élément qui sont automatiquement triées.

Exercice 6.6

- Écrire une fonction `coupe(L)` séparant les éléments d'une liste en deux listes de même taille, à un près. Cette fonction renverra le couple de ces sous-listes.
- Écrire une fonction `fusion(L1, L2)` qui prend en argument deux listes de tailles n1 et n2 triées par ordre croissant, et renvoie une liste de taille n1 + n2 contenant les éléments de `L1` et de `L2`, triés par ordre croissant.
- En déduire l'écriture d'une fonction récursive triant une liste par ordre croissant.
- En supposant que la taille de la liste s'écrit 2^p donner pour tout i le nombre d'appels à la fonction fusion entre des listes de taille 2^i et donner la complexité totale de ces appels. En déduire que ce tri a une complexité en $O(n\ln(n))$.
- Quelles sont les caractéristiques de ce tri

```python
def coupe(L):
    ''' sépare une liste en deux listes de même taille (à un élément près)
    retourne le tuple formé de ces deux listes'''
    L1 = []
    L2 = []
    for i in range(len(L)):
        if i % 2 == 0:
            L1.append(L[i])
        else:
            L2.append(L[i])
    return L1, L2

def coupe2(L):
    L1 = [L[i] for i in range(len(L)//2)]
    L2 = [L[i] for i in range(len(L)//2, len(L))]
    return L1, L2

def fusion(L1, L2):
    ''' L1 et L2 sont deux listes triées par ordre croissant
    retourne une liste L composée de tous les éléments de L1 et L2 triés par ordre croissants'''
    L = []
    i = 0
    j = 0
    while i < len(L1) and j < len(L2):
        if L1[i] < L2[j]:
            L.append(L1[i])
            i = i + 1
        else:
            L.append(L2[j])
            j = j + 1
    for k in range(i, len(L1)):
        L.append(L1[k])
    for k in range(j, len(L2)):
        L.append(L2[k])
    return L

def fusion_recursivite(L1, L2):
    if L1 == []:
        return L2
    if L2 == []:
        return L1
    if L1[0] < L2[0]:
        return [L1[0]] + fusion_recursivite(L1[1:], L2)
    else :
        return [L2[0]] + fusion_recursivite(L1, L2[1:])

def tri_fusion(L):
    if len(L) <= 1:
        return L
    else:
        L1, L2 = coupe(L) #L1, L2 = L[:len(L)//2], L[len(L)//2::]
        return fusion(tri_fusion(L1), tri_fusion(L2))
```

VII. Tri rapide

Le « tri rapide » est un autre algorithme de tri, utilisant la méthode « diviser pour régner » de même complexité que le tri fusion si on utilise de l'aléatoire. Son principe est d'effectuer les comparaisons au moment de la division plutôt qu'au moment des fusions. Plus précisément :

- Si la liste est vide ou admet un seul élément alors elle est triée ;
- sinon :
  - on définit le pivot p comme le dernier élément de la liste.
  - On divise la liste en 3 : les éléments strictement inférieurs à p, ceux égaux à p, et ceux strictement supérieurs. On trie récursivement la première et la dernière liste, et on concatène les résultats.

1. Appliquer à la main l'algorithme à la liste `L = [ 10, 3, 5, 6, 8, 12, 4, 7]`
2. Écrire une fonction `repartition` prenant en argument une liste et renvoyant les 3 listes décrites ci-dessus.
3. En déduire une implémentation du tri rapide.
4. Quelles sont les caractéristiques de ce tri ?
5. Avec notre choix de pivot, que se passe-t-il si la liste est triée ? que pourrait-on faire pour éviter cela ?

```
## Tri rapide
def repartition(L):
    p = L[len(L)-1]
    L1 = []
    L2 = []
    L3 = []

    for elt in L:
        if elt < p:
            L1.append(elt)
        elif elt == p:
            L2.append(elt)
        else:
            L3.append(elt)
    return (L1, L2, L3)

def tri_rapide(L):
    if len(L) <= 1:
        return L
    else :
        L1, L2, L3 = repartition(L)
        return tri_rapide(L1) + L2 + tri_rapide(L3)
```

```python
## Tri rapide en place
def repartition(L, a, b):
    p = L[b]
    ind_p = a
    for i in range(a, b):
        if L[i] <= p:
            echange(L, i, ind_p)
            ind_p += 1
    echange(L, b, ind_p)
    return ind_p

def tri_sous_liste(L, a, b):
    if b > a:
        ind_p = repartition(L, a, b)
        tri_sous_liste(L, a, ind_p - 1)
        tri_sous_liste(L, ind_p +1, b)

def tri_rapide(L):
    tri_sous_liste(L, 0, len(L) - 1)
```

VIII. Tri sans comparaison

Déjà vu dans le chapitre sur les dictionnaires.
Si on sait que les valeurs d'une liste sont des entiers pas trop grands, on peut trier la liste en comptant le nombre d'occurrences de chaque valeur, puis en réécrivant ensuite dans la liste autant d'occurrences de chaque valeur, par ordre croissant.

1. Écrire une fonction `tri_par_denombrement(l)` qui réalise cet algorithme.
2. Quelles sont les caractéristiques de ce tri

```python
def occurrences2(t):
    d = {}
    mini, maxi = t[0], t[0]
    for elt in t:
        if elt in d:
            d[elt] += 1
        else:
            d[elt] = 1
        mini = min(mini, elt)
        maxi = max(maxi, elt)
    return d, mini, maxi


def tri_dénombrement(l):
    nb, mini, maxi = occurrences2(l)
    res = []
    for i in range(mini , maxi + 1):
        if i in nb:
        # Il faut rajouter nb[i] fois i dans res.
            for k in range(nb[i]):
                res.append(i)
    return res
```