Chapitre 2 : Listes, dictionnaires, tris

1 Listes

Une liste est une séquence finie d'éléments, possiblement de types différents. La syntaxe consiste à les mettre entre crochets, séparés par des virgules. Voici un exemple : `[2,4.6,False]`.

1 A) Construction de listes

On peut construire une liste de plusieurs manières :

— par la donnée explicite des éléments, entre crochets, séparés par des virgules, comme ci-dessus.

— par concaténation de listes (à l'aide de `+`) : `[1,2,3]+[4,5,6]` s'évalue en `[1,2,3,4,5,6]`.

— `list(x)` permet de convertir un objet itérable en liste. Par exemple, `list(range(5))` s'évalue selon la liste `[0,1,2,3,4]` et `list("chose")` en `['c','h','o','s','e']`.

— par compréhension, très pratique avec la structure suivante : `L=[f(x) for x in iterable if P(x)]`, où `f(x)` est une expression dépendant (ou non) de `x`, et `P(x)` est une expression booléenne (facultative). Par exemple, `[x*x in range(5) if x%2==0]` s'évalue en la liste `[0,4,16]`.

1 B) Accès aux éléments

Pour une liste `L`, sa *longueur* (nombre d'éléments, length en anglais) est accessible avec `len(L)`. On notera que la longueur d'une liste est enregistré en mémoire et que donc l'appel à la fonction `len` se fait à coût constant (le nombre d'opérations effectuées ne dépend pas de la longueur de la liste).

En notant n cette longueur, les éléments sont indexés par les entiers de 0 à $n-1$. La commande `L[i]` permet d'accéder au i-ème élément de la liste `L`.

```
>>> L=list(range(3,9))
>>> len(L)
6
>>> L[3]
6
>>> L[len(L)-1]
8
```

Si on demande l'accès à un caractère d'indice négatif i compris entre -1 et $-n$, Python renvoie l'élément d'indice $n+i$. En particulier, `L[-1]` renvoie le dernier élément de la liste `L` :

```
>>> L[-1]
8
>>> L[-5]
4
```

Si on tente d'accéder à un indice qui n'appartient pas à $[\![-n, n-1]\!]$, cela produit une erreur.

```
>>> L[len(L)]
Traceback (most recent call last):

File "<ipython-input-52-8b7515244d2d>", line 1, in <module>
L[len(L)]

IndexError: list index out of range
```

Exercice 1.1 (Liste ordonnée)**.** Donner une fonction `Croissante(L)` renvoyant un booléen selon que la liste en entrée `L` est classée par ordre croissant ou non. Combien cette fonction effectue-t-elle d'opérations ?

Pour tester si un élément appartient à une liste (ou tout autre structure de données), on peut utiliser l'opérateur `in` (ou `not in`). Le résultat donne un booléen :

```
>>> 2 in [2,3,4]
True
>>> 2 not in (3,4,5)
True
>>> 'ab' in 'abcd'
True
```

1 C) Modification d'un élément

Étant donné une liste `L`, on peut modifier l'élément d'indice i de `L` en le remplaçant par l'élément de son choix. La syntaxe est la même que pour une affectation.

```
>>> L=list(range(1,6))
>>> L
[1,2,3,4,5]
>>> L[2]=10
>>> L
[1,2,10,4,5]
```

1 D) Slicing (tranchage)

On peut créer une nouvelle liste en extrayant certains éléments d'une liste. Pour extraire les éléments d'indice entre i inclus et $j \geq i$ exclu, on utilise `L[i:j]`. La liste obtenue est donc composée des éléments `L[i], L[i+1], ..., L[j-1]`. L'un ou l'autre de ces deux indices peut être omis. La liste extraite commence alors à `L[0]` si i est omis et finit à `L[-1]` si j est omis. Ce mécanisme est tolérant envers les indices trop grands ou trop petits (les indices négatifs sont cependant interprétés comme précédemment) et si $i \geq j$, on obtient la liste vide.

```
>>> L
[1,2,10,4,5]
>>> L[3:4]
[4]
>>> L[4:3]
[]
>>> L[:4]
[1,2,10,4]
```

```
>>> L[:8]
[1,2,10,4,5]
>>> L[-6:-3]
[1, 2]
>>> L[6:8]
[]
```

On peut également spécifier un pas, positif ou négatif. L'interprétation est la même que pour l'itérateur `range`.

```
>>> L[::2]
[1,10,5]
```

1 E) Méthodes sur les listes

Python est un langage *orienté objet*. À chaque classe d'objet telle que les listes, peuvent s'appliquer plusieurs *méthodes* qui modifient l'objet et renvoient certaines de ses caractéristiques. Deux d'entre elles sont au programme de CPGE : `append` et `pop`. Elles permettent respectivement d'ajouter ou d'enlever un élément en fin de liste.

En respectant la syntaxe général `objet.methode(parametres)`, on a donc :

— `L.append(x)` : ajoute `x` à la fin de `L` ;

— `L.pop()` : supprime le dernier élément de `L` et le renvoie.

La méthode `append` renvoie `None`, par contre elle modifie l'objet (la liste).

Exercice 1.2. Déclarer une fonction `nettoyage` qui prend en entrée une liste `L` et un objet `a`, puis renvoie la liste obtenue en retirant de `L` tous les termes égaux à `a`.

1 F) Listes et sémantique de pointeur

L'affectation d'une liste dans une variable est particulière comme le prouve l'exemple suivant :

```
>>> L1=[4,5,6]
>>> L2=L1
>>> L1[0]=9
```

```
>>> L1.append(22)
>>> print(L2)
[9,5,6,22]
```

Contrairement au cas des variables, la modification de `L1` a entraîné celle de `L2`. Lorsqu'on crée une liste, la variable utilisée est ce que l'on appelle une référence (ou un pointeur) vers l'emplacement mémoire où est stockée la liste. L'instruction `L2=L1` du premier exemple stocke dans la variable `L2` la référence stockée dans `L1`. Autrement dit, l'emplacement mémoire désigné par `L1` et `L2` est le même. Lorsqu'on effectue une instruction telle que `L1[0]=9` ou `L1.append(22)`, on va modifier directement la mémoire et ce changement sera donc visible lorsqu'on écrit `print(L2)`, car cette action va chercher en mémoire ce qu'indique `L2`.

Pour éviter ce genre d'erreurs, il faut procéder différemment :

```
>>> L1=[4,5,6]
>>> L2=L1[:]
>>> L1[0]=9
```

```
>>> L1.append(22)
>>> print(L2)
[4,5,6]
```

Dans le deuxième exemple, l'instruction `L2=L1[:]` permet de créer une liste dont les éléments sont les mêmes que ceux de `L1`, mais ailleurs en mémoire. Les références `L1` et `L2` pointent vers des endroits différents en mémoire, donc la modification de l'une laisse l'autre inchangée.

1 G) Listes de listes

Voici un exemple composé de listes de listes :

```
>>> L=[[0,1,2,3],[4,5]] # une liste de deux listes
>>> L[0] #premier élément de L
[0,1,2,3]
>>> L[1][0] #premier élément du deuxiéme élément de L
4
>>> L[0][2]=6
>>> print(L)
[[0,1,6,3],[4,5]]
```

1 H) Tuples

Un tuple (`tuple`) est très similaire à une liste, mais on ne peut ni modifier, ni ajouter, ni supprimer ses éléments. Cela correspond à la structure mathématique de n-uplet. On parle de structure immuable, statique, ou non mutable. En contrepartie de cette rigidité les tuples sont très compacts (ils occupent peu de mémoire) et l'accès à leurs éléments est plus rapide que pour une liste.

Pour la syntaxe, on crée un tuple en écrivant ses éléments, séparés par des virgules et encadrés par des parenthèses. On peut omettre les parenthèses si cela ne créé pas d'ambiguïté. Un tuple constitué d'un seul élément `a` doit s'écrire `a,` ou `(a,)`. Le tuple sans élément s'écrit `()`. Les opérations sur les listes sont valables sur les tuples :

```
>>> t=4, True, 0.5 ; v=(6,) ; w=((7,8),)
>>> t+v
(4, True, 0.5, 6)
>>> t[1:]
(True, 0.5)
```

Comme on l'observe dans cet exemple, un élément d'un tuple peut être un tuple à son tour. Attention cependant à l'immuabilité d'un tuple : on ne peut pas modifier un élément.

```
>>> t[0]=8
Traceback (most recent call last):

File "<ipython-input-7-be0ada286cc5>", line 1, in <module>
t[0]=8

TypeError: 'tuple' object does not support item assignment
```

1 I) Chaînes de caractères `str`

Une donnée de type chaîne de caractères est une suite de caractères quelconques. On crée une chaîne de caractères en écrivant les caractères en question soit entre apostrophes, soit entre guillemets : `'Salut'` et `"Salut"` sont deux écritures correctes de la même chaîne de caractère. La structure d'une chaîne de caractère est proche de celle d'un tuple. On retrouve la propriété d'immuabilité : on ne peut pas changer un caractère.

Comme pour les tuples, on peut concaténer deux chaînes de caractères à l'aide de `+` pour en produire une troisième, la longueur de la chaîne est donnée par `len` et l'accès aux caractères est similaire.

```
>>> "C'est une"+" phrase complete."
"C'est une phrase complete."
>>> a="Une chaine de caracteres"
>>> a[0] ; a[5] ; a [-1]
'U'
'h'
's'
>>> a[::2]
'Uecan ecrcee'
```

Attention, les chaînes de caractères sont *immuables* : une fois créées, on ne peut pas les modifier, ou leur rajouter des éléments. Il faut créer une nouvelle chaîne qu'on peut éventuellement réaffecter à la même variable.

```
>>> a='poulet'
>>> a[0]='b'
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> a='b'+a[1:]
>>> a
'boulet'
```

1 J) Tableaux

Pour manipuler des matrices avec Python, on utilise généralement le type `array`. Pour cela, on a besoin de charger le module `numpy`. On supposera qu'on a effectué pour cela la commande `import numpy as np`. Le type « tableau » est assez proche du type « liste » en Python mais ne se manipule pas la même manière.

En Python, pour créer la matrice $A = \begin{pmatrix} 26 & 7 & 85 \\ 1.5 & 4 & -2 \end{pmatrix}$ avec un tableau on utilisera la commande :

```
A = np.array([[26,7,85],[1.5,4,-2]])
```

Contrairement aux matrices en mathématiques, les lignes et les colonnes sont numérotées à partir de 0 dans un tableau `numpy` à l'image des éléments d'une liste.

Attention, contrairement aux « listes de listes », un tableau `array` comprend obligatoirement des objets d'un même type et toutes les lignes (respectivement toutes les colonnes) contiennent le même nombre d'éléments.

Il existe des commandes Python du module `numpy` pour créer la matrice nulle et la matrice identité. Les commandes : `np.zeros((2,4))` et `np.eye(3)` créent respectivement la matrice nulle à 2 lignes et 4 colonnes et la matrice identité à 3 lignes et 3 colonnes.

Pour créer une matrice diagonale, on peut effectuer : `np.diag([2,4,5])` qui renvoie une matrice diagonale à 3 lignes et 3 colonnes et qui comporte 2, 4, 5 sur la diagonale.

Pour additionner deux matrices, on utilisera l'opérateur `+` et pour multiplier par un scalaire, on utilise `s*A` où `s` est le scalaire et `A` la matrice. Pour effectuer le produit de deux matrices, on utilise la commande `np.dot(A,B)`, pour calculer AB.

2 Dictionnaires

Les dictionnaires en Python (appelés aussi tableaux associatifs ou table de hachage), permettent d'associer des valeurs à des clés. A partir d'une clé, on peut accéder à la valeur qui lui est associée. Les dictionnaires sont des conteneurs comme les listes et les tuples. Ils sont mutables : on peut ajouter, supprimer ou modifier le contenu.

Les dictionnaires ne sont pas des séquences : on ne peut pas accéder à leur contenu via un indice. L'ordre des éléments dans un dictionnaire n'a pas d'importance. Les clés peuvent être de type `str`, `int`, `float`, `tuple` de nombre, `tuple` de `tuple` de nombre... mais pas de type list ou un tuple de liste. Les valeurs peuvent être de n'importe quel type. Les dictionnaires sont de type `dict`.

2 A) Création d'un dictionnaire

Pour créer un dictionnaire vide, on indique `mon_dictionnaire={}`. On peut ensuite ajouter les valeurs avec les clés souhaitées : `mon_dictionnaire[cle]=valeur` ou, en une seule fois :

`mon_dictionnaire={clé1:valeur1,clé2:valeur2,...}.`

On peut également utiliser la syntaxe suivante : `dictionnaire=dict(clé1=valeur1,clé2,valeur2,...)`.

Remarque 2.1. La fonction dict() permet aussi de convertir une liste de listes à 2 éléments en dictionnaire.

Exemple 2.2. Voici un exemple complet :

```
# dictionnaire vide
dico_en_fr={}

# ajout des cles et des valeurs
dico_en_fr["yes"]="oui"
dico_en_fr["no"]="non"
dico_en_fr["why"]="pourquoi"

# définition directe d'un dictionnaire
dico_es_fr={"si":"oui","hoy":"aujourd'hui","porque":"pourquoi"}

# définition d'un dictionnaire en utilisant la fonction dict
dico_en_fr_nb=dict(one=1,two=2,three=3)
```

```
# transformation d'une liste de tuple á 2 éléments
liste_es_nb=[("uno",1),("dos",2),("tres",3)]
dico_es_nb=dict(liste_es_nb)

# définition d'un dictionnaire par compréhension
suite_carre={x:x**2 for x in range(5)}
```

2 B) Parcours d'un dictionnaire, accès aux clés et aux valeurs

Les méthodes `keys` et `values` donnent accès aux clés ou au valeurs. La méthode items donne accès à l'ensemble des couples.

En reprenant l'exemple précédent, la méthode `keys` donne accès aux clés. La commande `in` permet de tester l'appartenance d'une clé à un dictionnaire mais pas d'une valeur. La variable d'itération d'une boucle for sur un dictionnaire est une clé.

```
>>> dico_en_fr.keys()
dict_keys(['yes', 'no', 'why'])
>>> "yes" in dico_en_fr
True
>>> for objet in dico_es_fr:
                print(objet)
si
hoy
porque
```

L'instruction `valeur=dictionnaire[cle]` permet d'obtenir la valeur du dictionnaire correspondante à la clé. Si la clé n'existe pas, une erreur arrête le programme.

Remarque 2.3. Nous verrons dans le chapitre sur la programmation dynamique que le domaine des clés utilisables est limité. Ceci fera intervenir la notion de fonction de hachage.

2 C) Modification des données

Le contenu d'un dictionnaire peut être modifié en remplaçant la valeur associée à une clé par une autre valeur. On utilise pour cela la syntaxe d'affectation : `dictionnaire[clé]=nouvelle_valeur`.

Remarque 2.4. Si clé existe, sa valeur sera modifiée en `nouvelle_valeur`. Si clé n'existe pas, un nouvel élément est crée dans le dictionnaire.

```
>>> dico_es_nb["cuatro"]=5
>>> dico_es_nb
{'uno': 1, 'dos': 2, 'tres': 3, 'cuatro': 5}
>>> dico_es_nb["cuatro"]=4
>>> dico_es_nb
{'uno': 1, 'dos': 2, 'tres': 3, 'cuatro': 4}
```

La fonction `len()` renvoie la longueur d'un dictionnaire. On peut supprimer une entrée du dictionnaire avec l'instruction `del()`.

```
>>> len(dico_es_nb)
4
>>> del(dico_en_fr["no"])
>>> dico_en_fr
{'yes': 'oui', 'why': 'pourquoi'}
```

Exercice 2.5. Écrire une fonction Python `occurences` qui prend en entrée une chaîne de caractères et renvoie un dictionnaire dont les clés sont les lettres qui composent la chaîne de caractères et les valeurs le nombre d'apparition de chaque lettre. On considère que les chaînes de caractères sont uniquement composées des caractères de la chaîne `abcdefghijklmnopqrstuvwxyz` (pas de caractères accentués notamment. Par exemple, `occurences('endomorphisme')` renvoie :

`{'e': 2, 'n': 1, 'd': 1, 'o': 2, 'm': 2, 'r': 1, 'p': 1, 'h': 1, 'i': 1, 's': 1}`

2 D) Sémantique de pointeur

Les comportements sont similaires à ceux rencontrés avec les listes en ce qui concerne la copie d'un dictionnaire. Pour éviter ceci, on peut utiliser la méthode `copy`.

```
>>> dico = {"cle1":"valeur1","cle2":"valeur2"}
>>> copie1=dico.copy()
>>> copie2=dico
>>> copie2["test"]=4
>>> copie1
{'cle1': 'valeur1', 'cle2': 'valeur2'}
>>> dico
{'cle1': 'valeur1', 'cle2': 'valeur2', 'test': 4}
```

3 Tris

On distinguera deux approches pour nos algorithmes de tri :

— les tris dits « fonctionnels » : ce sont des algorithmes qui prennent en entrée un objet à trier et renvoient en sortie une image triée de l'objet, sans modifier l'objet d'origine.

— les tris dits « en place » : ce sont des algorithmes qui prennent en entrée un objet à trier et qui le trient par effet de bord en le modifiant sans créer d'image de cet objet.

Les algorithmes en place ont l'avantage d'utiliser moins d'espace mémoire mais modifient l'objet d'origine. Suivant les situations ou les demandes, on pourra préférer un algorithme fonctionnel ou un algorithme en place.

On gardera en fil-rouge l'exemple de la liste suivante : `L = [10, 85, 79, 2, 17, 25, 73, 12, 7]` que l'on souhaite trier dans l'ordre croissant pour obtenir `[2, 7, 10, 12, 17, 25, 73, 79, 85]`.

3 A) Tri par sélection

Le principe récurrent est de sélectionner l'élément (un des éléments) le(s) plus petit(s), de le placer en première position de la liste à trier et de considérer qu'il reste à ranger les éléments à partir de la deuxième position. Dans l'approche « fonctionnelle » on procède par empilement des valeurs sélectionnées dans une nouvelle structure créée pour recevoir les valeurs. Dans l'approche « en place » il faut choisir une façon de mettre à la bonne place l'élément sélectionné.

Simulation : états successifs, approche fonctionnelle

[2	.	.	.	.	.	.	.	.]
[2	7	.	.	.	.	.	.	.]
[2	7	10	.	.	.	.	.	.]
[2	7	10	12	.	.	.	.	.]
[2	7	10	12	17	.	.	.	.]
[2	7	10	12	17	25	.	.	.]
[2	7	10	12	17	25	73	.	.]
[2	7	10	12	17	25	73	79	.]
[2	7	10	12	17	25	73	79	85]

Simulation : états successifs, approche avec échanges

[10	85	79	2	17	25	73	12	7]
[2	85	79	10	17	25	73	12	7]
[2	7	79	10	17	25	73	12	85]
[2	7	10	79	17	25	73	12	85]
[2	7	10	12	17	25	73	79	85]
[2	7	10	12	17	25	73	79	85]
[2	7	10	12	17	25	73	79	85]
[2	7	10	12	17	25	73	79	85]
[2	7	10	12	17	25	73	79	85]

On donne une version fonctionnelle à gauche et une version avec échange à droite :

```
def TriSelectionF(L):
    n = len(L)
    cible = []
    aplacer = [True]*n
    for a in range(n):
        c = 0
        while not aplacer[c]:
            c += 1
        for b in range (c+1,n):
            if aplacer[b] and L[b] < L[c]:
                c = b
        cible.append(L[c])
        aplacer[c] = False
    return(cible)
```

3 B) Tri par insertion

Le principe est de ranger à sa place successivement chaque nouvel élément envisagé par rapport aux précédents. C'est le tri qu'on utilise pour ranger des livres sur une étagère, des cartes à jouer...

Simulation : états successifs, approche fonctionnelle

[.	.	.	.	.	.	.	.	.]
[10	.	.	.	.	.	.	.	.]
[10	85	.	.	.	.	.	.	.]
[10	79	85	.	.	.	.	.	.]
[2	10	79	85	.	.	.	.	.]
[2	10	17	79	85	.	.	.	.]
[2	10	17	25	79	85	.	.	.]
[2	10	17	25	73	79	85	.	.]
[2	10	12	17	25	73	79	85	.]
[2	7	10	12	17	25	73	79	85]

Simulation : états successifs, approche avec échanges

[10	85	79	2	17	25	73	12	7]
[10	85	79	2	17	25	73	12	7]
[10	85	79	2	17	25	73	12	7]
[10	79	85	2	17	25	73	12	7]
[2	10	79	85	17	25	73	12	7]
[2	10	17	79	85	25	73	12	7]
[2	10	17	25	79	85	73	12	7]
[2	10	17	25	73	79	85	12	7]
[2	10	12	17	25	73	79	85	7]
[2	7	10	12	17	25	73	79	85]

Toute la difficulté (ou justement l'absence de difficulté) réside dans l'insertion dans une structure **déjà triée**. On donne une version fonctionnelle à gauche et une version avec échange à droite :

```
def TriInsertionF(L):
    cible = L[:1]
    for e in L[1:]:
        r = 0
        while r < len(cible) and cible[r] < e:
            r += 1
        cible = cible[:r] + [e] + cible[r:]
    return cible
```

```
def TriInsertionEP(L):
    n = len (L)
    for k in range(1,n):
        temp = L[k]
        r = k
        while r > 0 and L[r-1] > temp:
            L[r]=L[r-1]
            r = r-1
        L[r] = temp
```

3 C) Tri fusion

Nous allons créer un programme récursif utilisant la méthode « diviser pour régner ».

On coupe le tableau en deux morceaux toujours égaux (à un élément près) et on répète l'opération sur les sous-tableaux jusqu'à obtenir des tableaux de longueur 0 ou 1. Il reste ensuite l'étape délicate appelée fusion. Elle consiste à fusionner deux tableaux triés de telle sorte qu'on obtienne un tableau trié.

Simulation : états successifs du tableau argument :

[10	85	79	2]	[17	25	73	12	7]
[10	85]	[79	2]	[17	25]	[73	12	7]
[10]	[85]	[79]	[2]	[17]	[25]	[73]	[12	7]
[10]	[85]	[79]	[2]	[17]	[25]	[73]	[12]	[7]
[10	85]	[2	79]	[17	25]	[73]	[7	12]
[2	10	79	85]	[17	25]	[7	12	73]
[2	10	79	85]	[7	12	17	25	73]
[2	7	10	12	17	25	73	79	85]

La fonction principale est la fonction `fusion` qui permet de « fusionner » deux tableaux **triés** en un tableau **trié**. Pour cela on procède de manière récursive : le plus petit élément des deux tableaux est placé en première position du nouveau tableau, il reste ensuite à fusionner les deux tableaux privés de cet élément.

On commence donc par écrire la fonction `fusion` qui concatène deux tableaux triés `tG` et `tD` en un seul tableau trié composé des éléments de `tG` et `tD` où on donne une version récursive et une version itérative :

```
def fusion(tG,tD):
    L = []
    if tG == []:
        return tD
    elif tD == []:
        return tG
    else:
        if tG[0]<tD[0]:
            L.append (tG[0])
            L += fusion(tG[1:], tD)
        else:
            L.append (tD[0])
            L += fusion (tG, tD[1:])
    return L
```

```
def fusion (tG, tD) :
        L = []
        nG = len (tG)
        nD = len (tD)
        iG = 0
        iD = 0
        while iG < nG and iD < nD :
        if tG[iG] < tD[iD] :
                L.append (tG[iG])
                iG += 1
        else :
                L.append (tD[iD])
                iD += 1
        return L + tG[iG:] + tD[iD:]
```

Par exemple, `fusion([1,3],[2,4])` renvoie `[1,2,3,4]`.

On peut désormais écrire la fonction de tri sous forme récursive en coupant en deux le tableau à chaque étape :

```
def triFusion(liste):
    n = len(liste)
    if n <= 1:
        return liste
    else:
        return fusion (triFusion(liste[:n//2]),triFusion(liste[n//2:]))
```

Commentaire :

3 D) Tri rapide

L'idée générale est à nouveau un partage en deux analogue au tri par fusion mais au lieu d'une partition dichotomique systématique du tableau, qui oblige à une fusion délicate et à l'emploi de listes temporaires, le tri rapide partitionne le tableau autour d'une valeur charnière - le pivot noté x plus bas - et trie en place les deux parties produites. Pour le choix de la valeur charnière, plusieurs stratégies sont possibles, puisque la complexité résultera en partie de ce choix, par exemple : le premier élément, un élément tiré au sort, etc. On crée alors deux sous-tableaux : un contenant les éléments strictement inférieurs à x, et l'autre contenant les éléments supérieurs ou égaux à x. On relance le tri de manière récursive sur ces deux sous-tableaux qui deviennent donc des listes triées, et il ne reste plus qu'à les concaténer, en intercalant le pivot entre les deux. En effet, on remarque qu'on reforme ainsi la liste initiale mais sous forme triée.

On procède ensuite comme dans le tri fusion en créant une fonction récursive utilisant le découpage précédent. Cet algorithme est un nouvel exemple de la méthode générale utilisée en récursivité *diviser pour régner* : on divise le problème initial en deux sous-problèmes que l'on sait résoudre, et on revient ensuite au problème initial. Dans le cas du tri rapide, les appels récursifs sont lancés jusqu'à tomber sur des listes de taille 1 ou 0, donc déjà triées. Il ne reste plus qu'à les concaténer dans le bon ordre.

On donne ici un algorithme dans le cas où le pivot choisi est le premier élément.

Simulation : états successifs du tableau argument :

[10	85	79	2	17	25	73	12	7]
[2	7	10	85	79	17	25	73	12]
[2	7	10	79	17	25	73	12	85]
[2	7	10	17	25	73	12	79	85]
[2	7	10	12	17	25	73	79	85]
[2	7	10	12	17	25	73	79	85]
[2	7	10	12	17	25	73	79	85]

On commence par créer une fonction `partition` qui étant donné une liste `L` et deux indices a et b, partitionne la liste `L[a:b]` suivant le pivot `L[a]` et renvoie la nouvelle position du pivot.

```
def partition (L,a,b):
    pivot = L[a]
    pos=a
    for i in range(a+1,b):
        if L[i]< pivot:
            L[i],L[pos+1] = L[pos+1],L[i]
            pos += 1
    if pos != a:
        L[a],L[pos] = L[pos],L[a]
    return pos
```

```
def TriRapide(L,a,b):
    if a<b:
        position = partition (L,a,b)
        TriRapide(L,a, position )
        TriRapide(L, position +1,b)

def TriRapideEP(L):
    TriRapide(L,0,len(L))
```

On se sert ensuite de cette fonction dans une fonction récursive de telle sorte que la fonction `TriRapideEP` trie la liste en place.
Commentaires :

3 E) Complexités des algorithmes de tri

Pour deux données de même taille, un algorithme n'effectue pas nécessairement le même nombre d'opérations élémentaires. Par exemple, considérons la fonction python suivante qui détermine si un objet x est présent dans une liste L.

```
def appartenance(L,x):
    trouve=False
    n=len(L)
    k=0
    while not(trouve) and k<n:
        trouve=(L[k]==x)
        k+=1
    return(trouve)
```

```
>>> appartenance([1,6,7,8,-3],-3)
True
```

Dans cet exemple, la taille du problème est le nombre d'éléments dans la liste L. Les opérations pertinentes à compter sont les comparaisons entre x et le k-ème élément de L. Comme la boucle s'arrête si l'élément est trouvé dans la liste, le nombre de comparaisons effectuées peut varier de 1 (si x est le premier élément de la liste) à n (si x est le dernier élément de la liste ou bien si x n'est pas dans la liste).
On est donc amené à considérer la complexité dans le pire ou le meilleur cas. Ceci fournit donc un encadrement de la complexité.

Exemple 3.1. Déterminer les complexités $C_+(n)$ et $C_-(n)$ dans le pire et le meilleur cas de l'algorithme du tri par insertion dans sa version en place où n est la taille de la liste en entrée.

Théorème 3.2. Notons n la longueur de la liste triés. Les complexités temporelles dans le pire et le meilleur cas des algorithmes de tri sont sont données par le tableau :

Nom du tri	meilleur cas	pire cas
Tri par insertion	n	n^2
Tri par sélection	n^2	n^2
Tri fusion	$n\ln(n)$	$n\ln(n)$
Tri rapide	n	n^2

Démonstration de la complexité temporelle pour le tri fusion :

3 F) Choix aléatoire du pivot pour le tri rapide : une approche de la complexité moyenne (*hors-programme*)

On change désormais de stratégie pour le choix du pivot en considérant le code suivant :

```
from random import randint
def partition(L,a,b):
    pivot=randint(a,b)
    L[b],L[pivot]=L[pivot],L[b]
    newPivotIndice=a-1
    for index in range(a,b):
        if L[index]<L[b]:
            newPivotIndice=newPivotIndice+1
            L[newPivotIndice],L[index]=L[index],L[newPivotIndice]
    L[newPivotIndice+1],L[b]=L[b],L[newPivotIndice+1]
    return newPivotIndice+1

def TriRapide(L,a,b):
    if a<b:
        p=partition(L,a,b)
        TriRapide(L,a,p-1)
        TriRapide(L,p+1,b)

def TriRapideEP(L):
    TriRapide(L,0,len(L)-1)
```

On ne s'attachera pas ici à la complexité ni au fonctionnement de `randint(a,b)` qui renvoie un élément un nombre entier entre a et b « au hasard » : on considère que sa complexité est un $O(1)$. Pour étudier la complexité moyenne, on va utiliser des résultats du cours de probabilités. On note $a_1, \dots, a_n$ les éléments du tableau donnés par ordre croissant. Pour $i < j$, on note $A_{i,j} = \{a_i, a_{i+1}, \dots, a_j\}$. Initialement, le tableau à trier est $a_{\sigma(1)}, \dots, a_{\sigma(n)}$ où σ est une permutation de $[\![1, n]\!]$.

Les opérations à effectuer sont les choix de pivot et les comparaisons qui ont lieu lors des partitions autour de chaque pivot. Dès lors qu'un pivot est choisi il n'est plus impliqué dans les partitions qui suivent et donc une exécution de l'algorithme fait intervenir au plus n choix de pivot, et chaque choix se fait en temps constant (c'est ce qu'on a supposé plus haut). De plus, tout élément est comparé à un élément au plus une fois, car tout élément n'est comparé qu'à un pivot, et une fois effectuée la partition autour d'un pivot donné, celui-ci n'intervient plus dans l'exécution de l'algorithme une fois la partition finie. En notant X le nombre total de comparaison on obtient que la complexité d'une exécution est donc $O(X + n)$ (X dépendant de n). De plus si l'on note $X_{i,j}$ la variable aléatoire qui vaut 1 si a_i est comparé à a_j et 0 sinon, on déduit que l'on a pour une exécution donnée

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{i,j}.$$

X est une variable aléatoire et le calcul de son espérance $E(X)$ donnera la complexité moyenne de l'algorithme.

Or a_i est comparé avec a_j si et seulement si le premier élément de $A_{i,j}$ à être choisi comme pivot est soit a_i soit a_j. En effet si c'est un autre élément qui est choisi en premier comme pivot, alors a_i et a_j sont envoyés dans deux sous-tableaux différents de la partition, et les comparaisons sont effectuées exclusivement entre éléments d'un même sous-tableau. De plus la probabilité que le premier élément de $A_{i,j}$ à être choisi comme pivot soit a_i ou a_j vaut

$$\frac{\text{card}(\{a_i, a_j\})}{\text{card}(A_{i,j})} = \frac{2}{i - j + 1}$$

car on peut faire l'hypothèse que les choix de pivot sont équiprobables. Ainsi, $X_{i,j}$ suit une loi de Bernoulli de paramètre $\frac{2}{i-j+1}$ ce qui donne :

$$E(X) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E(X_{i,j}) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{i - j + 1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \leqslant \sum_{i=1}^{n} \sum_{k=1}^{n} \frac{2}{k} \leqslant 2n\big(\ln(n) - \ln 2 + 1\big)$$

où la dernière inégalité vient d'une comparaison série intégrale. On obtient bien un $O(n \ln n)$. On a donc obtenu un tri qui évite la complexité spatiale tout en conservant la complexité temporelle **moyenne** du tri fusion.

4 Exercices

Exemple 4.1 (Matrices et dictionnaires)**.**

1. Comment peut-on manipuler des matrices en Python en utilisant un dictionnaire ?

On s'intéresse ici aux matrices parcimonieuses, c'est-à-dire dont la plupart des coefficients sont nuls. Une telle matrice M de dimensions (n, p) pourra être codée par un dictionnaire ayant pour couples clefs/valeurs :

- `'dim'` : (n, p)
- (i, j) : M_{ij} pour chaque couple (i, j) tel que $M_{i,j} \neq 0$.

2. Donner le dictionnaire qui code la matrice : $\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 \end{pmatrix}$.
3. Proposer une fonction d'addition de deux matrices de même dimension.
4. Si la première matrice contient c coefficients non nuls et la seconde c', quelle est la complexité temporelle de cet algorithme ?

Exemple 4.2 (Tri par dénombrement)**.** Soit N un entier naturel non nul. On cherche à trier une liste `L` d'entiers naturels strictement inférieurs à N.

1. Écrire une fonction `comptage`, d'arguments `L` et `N`, renvoyant une liste `P` dont le k-ième élément désigne le nombre d'occurrences de l'entier k dans la liste `L`.
2. Utiliser la liste `P` pour en déduire une fonction tri, d'arguments `L` et `N`, renvoyant la liste `L` triée dans l'ordre croissant.
3. Quelle est la complexité temporelle de cet algorithme ? La comparer à la complexité d'un tri par insertion ou d'un tri fusion.