

# Thèmes d'informatique en Python

Mickaël Péchaud

2018



# Table des matières

<b>1</b>	<b>Traitement simple d'images</b>	<b>5</b>
1.1	Les images en Python avec <i>PIL</i> . . . . .	5
1.2	Un exemple simple . . . . .	6
1.3	Exercices . . . . .	7
<b>2</b>	<b>Algorithmes sur une table de données</b>	<b>15</b>
2.1	Recherche dans une telle base . . . . .	16
2.2	Un problème de temps d'exécution . . . . .	17
2.3	Recherche de doublons . . . . .	22
2.4	Trier des données . . . . .	23
2.4.1	Tri par sélection . . . . .	24
2.4.2	Tri par insertion . . . . .	25
2.4.3	Tri fusion . . . . .	26
2.5	Persistance des données . . . . .	27
<b>3</b>	<b>Principes (très) généraux du fonctionnement d'un moteur de recherche</b>	<b>29</b>
3.1	Indexation . . . . .	29
3.2	Calcul de popularité sur un exemple jouet . . . . .	33
<b>A</b>	<b>Complexité</b>	<b>37</b>

## Préambule

Ce document est distribué sous licence CC BY-NC-SA 3.0 FR.

<https://creativecommons.org/licenses/by-nc-sa/3.0/fr/>

Vous pourrez retrouver ce poly, son corrigé, ainsi que les fichiers *Python* à l'adresse suivante :

<http://mpechaud.fr/formationpython>

Pour toutes remarques, questions, merci de vous adresser par mail à l'auteur de ce poly (prenom@protonmail.com) !

Ce poly se base sur les connaissances acquises principalement dans la *Partie 1* de la formation *Python*.

Il vise à introduire des thèmes dont il est question dans les projets de programmes de *Sciences numériques et technologie* (classe de seconde, enseignement commun) et *Numérique et sciences informatiques* (classe de première, spécialité).



# Chapitre 1

## Traitement simple d'images

Dans toute cette section, on travaillera dans le fichier *imagesbases.py* fourni. Les corrections des exercices sont en bas de ce fichier.

### 1.1 Les images en Python avec *PIL*

Il existe plusieurs bibliothèques Python permettant de manipuler les images. Nous allons utiliser *Python Image Library* (PIL pour les intimes – en fait sa branche *Pillow* qui est encore activement maintenue à l'heure actuelle). Voici les commandes qui nous seront utiles :

- `from PIL import Image` importe une fois pour toute les fonctions suivantes.
- `im = Image.open('fichier.jpg')` charge une image<sup>1</sup>.
- `im.save('fichier.jpg')` sauvegarde une image.
- `im.show()` affiche une image.
- `im.width` et `im.height` renvoient la largeur et la hauteur de l'image en pixels.
- `im.mode` contient des informations sur le type d'image chargée. Il s'agit d'une chaîne de caractères pouvant prendre entre autres l'une des valeurs suivante :

**L** : image en niveaux de gris. Chaque pixel est codé par un entier sur 8 bits, donc entre 0 et 255.

**RGB** : image couleur. Chaque pixel est codé par un triplet d'entiers, chacun étant codé sur 8 bits, donc entre 0 et 255. Ces trois entiers donnent respectivement les quantités de rouge, vert et bleu (en synthèse additive des couleurs) présentes sur ce pixel (on parle de *RVB* en français).

**RGBA** : comme **RGB**, mais avec sur chaque pixel une 4<sup>ème</sup> composante correspondant à la transparence (ou *canal alpha*), codée par un entier sur 8 bits : 255 correspond à un pixel totalement opaque, et 0 à un pixel totalement transparent.

**HSV** : chaque pixel est représenté par trois composantes – chacune étant codée sur 8 bits :

**Hue** : La *teinte* code la couleur en suivant le cercle chromatique (0 pour rouge, 43 pour jaune...)

**Saturation** : La *saturation* représente l'intensité de la couleur. Plus elle est proche de 0, plus la couleur se rapproche d'un gris.

**Value** : La *valeur* représente la brillance de la couleur (0 correspond à noir).

Il s'agit d'un changement de repère par rapport au codage *RGB*. On parle de *TSV* en français. La fonction `testHSV` permet de se donner une intuition de tout cela.

- `r=Image.new("L", (200,150))` créé une nouvelle image noire, de largeur 200 et de hauteur 150. Le mode, indiqué par le "L" peut prendre l'une des valeurs indiquées ci-dessus – ici, il s'agit donc d'une image en niveaux de gris.

---

1. Il faut pour cela que le *répertoire de travail* soit le répertoire contenant ce fichier. Pour voir ce répertoire, après avoir importé le module `os` (`import os`), il faut utiliser la commande `os.getcwd()` (pour **c**urrent **w**orking **d**irectory). Pour le modifier, on utilise `os.chdir('/chemin/vers/le/nouveau/repertoire')`.

- `c=im.getpixel((x,y))` renvoie la valeur du pixel de coordonnées entières  $(x, y)$  de l'image `im`. Pour une image en niveaux de gris, le résultat renvoyé est un entier compris entre 0 (noir) et 255 (blanc). Pour une image en *RGB*, ce sera un triplet d'entiers.
- `im.putpixel((x, y), c)` met à  $c$  (dont le type dépend du type d'image) le pixel  $(x, y)$  de l'image `im`.

### Exercice 1 – Évaluation de la taille mémoire d'une image

1. Quel est l'espace mémoire nécessaire pour stocker une image couleur (sans canal alpha) de  $3200 \times 2400$  pixels ? On supposera comme ci-dessus que chacun des canaux de couleur est codé sur 8 bits.

Chaque canal de couleur est codé sur 8 bits, i.e. 1 octet. On a donc besoin de 3 octets par bits, et donc de  $3200 \times 2400 \times 3$  octets pour l'image, c'est à dire environ *20Mo*. C'est un ordre de grandeur raisonnable pour la taille d'une image brute (format de type *raw*) sortant d'un appareil photo.

2. Une taille typique d'une image sur une page web est 500ko. Quel est le nombre maximal de pixels sur le côté d'une image carrée que l'on va pouvoir stocker comme à la question précédente sur 500ko ?

En reprenant les calculs ci-dessus, et en notant  $n$  le nombre de pixels sur un côté de l'image, on doit avoir  $3c^2 \leq 500.10^3$ , ce qui donne la valeur maximale  $n = 408$ .

3. En pratique si l'on prend une image au format *jpeg* d'environ 500ko, on observe typiquement qu'elle est constituée de  $1200 \times 1600$  pixels. Comment est-ce possible ?

Pour coder dans un format brut une image  $1600 \times 1200$ , il faut environ 5Mo en reprenant les calculs ci-dessus.

Le format *jpeg* – ainsi que de nombreux autres formats image – utilise des algorithmes de compression avec perte. Au prix d'une perte de la qualité de l'image, on diminue fortement sa taille, ce qui est important pour l'encombrement mémoire, mais encore plus pour la transmission de cette image via un réseau comme internet.

À noter que dans le cas de *jpeg*, l'algorithme de compression utilise des variantes de transformées de Fourier – et que le même type de méthodes sont utilisées en compression video ou son (par exemple dans les formats son *ogg* ou *mp3* – par opposition à un format brut tel que *wav*).

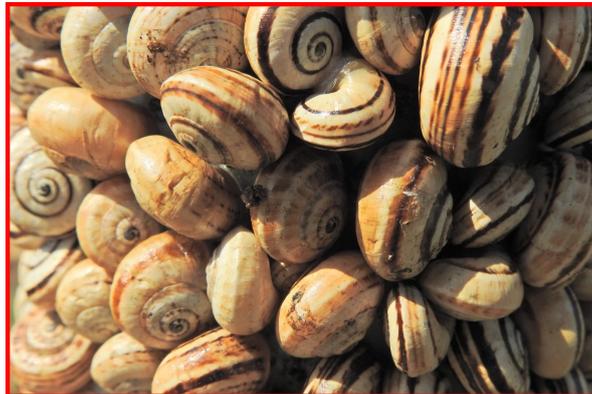
## 1.2 Un exemple simple

La fonction suivante charge une image couleur, met en rouge les pixels situés à 10 pixels ou moins du bord pour créer une bordure, puis sauvegarde l'image.



Code Python :

```
def contourRouge(imagein, imageout) :
    '''charge l'image imagein, créé une bande rouge de 10 pixels,
    puis sauvegarde dans imageout'''
    im = Image.open(imagein)
    if im.mode != 'RGB' :
        im = im.convert('RGB') #convertit l'image en RGB le cas échéant
    w = im.width
    h = im.height
    for i in range(w) :
        for j in range(10) :
            im.putpixel((i,j),(255,0,0)) #(255, 0, 0) = rouge
            im.putpixel((i,h-1-j),(255,0,0))
    for j in range(h) :
        for i in range(10) :
            im.putpixel((i,j),(255,0,0))
            im.putpixel((w-1-i,j),(255,0,0))
    im.save(imageout)
```



(a) Une image avec un contour rouge ajouté

## 1.3 Exercices

Une partie des exercices proposés est directement adaptée du projet de programme de SNT de seconde – section « Photographie Numérique ». Ce thème me paraît bien pouvoir se traiter à de petits projets, et le côté visuel des résultats obtenus est souvent motivant. Les différentes fonctions pourront être testées sur les images `escargots.JPG` (image couleur) et `escargotsNB.JPG` (en niveaux de gris) fournies.

**Exercice 2** Écrire une fonction qui « binarise » une image en niveau de gris avec un seuil  $s$  et affiche le résultat : tous les pixels de valeur inférieure ou égale à  $s$  deviennent noirs, et tous ceux de valeur strictement supérieure à  $s$  deviennent blancs.



Code Python :

```
def binarise(imagein, seuil) :
    '''charge l'image en niveaux de gris contenue dans
    le fichier imagein, la binarise avec le seuil
    indiqué, et affiche l'image ainsi obtenue'''
    im = Image.open(imagein)
    for i in range(im.width) :
        for j in range(im.height) :
            if im.getpixel((i, j)) <= seuil :
                im.putpixel((i, j), 0)
            else :
                im.putpixel((i, j), 255)
    im.show()
```

**Exercice 3** Écrire des fonctions qui transforment une image couleur codée en RGB en image en niveau de gris, et affiche cette dernière. On pourra pour cela tester plusieurs procédés :

1. Faire la moyenne des niveaux de rouge, vert et bleu pour obtenir un niveau moyen.



Code Python :

```
def conversionNB(imagein) :
    '''charge l'image couleur contenue dans le fichier
    imagein, la convertit en niveaux de gris (en
    utilisant une moyenne) et affiche l'image obtenue'''
    im = Image.open(imagein)
    r = Image.new('L', (im.width, im.height))
    for i in range(im.width) :
        for j in range(im.height) :
            valeur = im.getpixel((i, j))
            moyenne = (valeur[0]+valeur[1]+valeur[2])//3
            r.putpixel((i, j), moyenne)
    r.show()
```

2. Le procédé précédent ne prend pas en compte le fait que l'oeil humain n'a pas la même sensibilité à la luminance du rouge, du vert et du bleu. Dans l'industrie audiovisuelle, on reconstitue le niveau de gris en utilisant la formule  $0,2126R + 0,7152V + 0,0722B$ , où  $R$ ,  $V$  et  $B$  correspondent respectivement aux niveaux de rouge, vert et bleu (cf la recommandation *Rec. 709*). Coder, et tester ! En ajustant ces coefficients, on peut simuler l'effet de filtres utilisés en photographie noir et blanc.



Code Python :

```
def conversionNB2(imagein, lr, lg, lb) :
    '''charge l'image couleur contenue dans le fichier
    imagein, la convertit en niveaux de gris (en
    utilisant une moyenne pondérée par les coefficients
    lr, lg et lb) et affiche l'image obtenue'''
    im = Image.open(imagein)
    r = Image.new('L', (im.width, im.height))
    for i in range(im.width) :
        for j in range(im.height) :
            val = im.getpixel((i, j))
            moyenne = floor((lr*val[0]+lg*val[1]+lb*val[2]))
            r.putpixel((i, j), moyenne)
    r.show()
```

3. Convertir l'image RGB en HSV grâce à la commande `convert` utilisée dans l'exemple ci-dessus, mettre la saturation (S) à 0 (les concepteurs du programme ont semble-t-il oublié de tester cette méthode, qui est tentante sur le papier, mais qui ne fonctionne pas correctement, toute couleur vive étant convertie en blanc...).



Code Python :

```
def conversionNB3(imagein) :
    '''charge l'image couleur contenue dans le fichier
    imagein, la convertit en niveaux de gris (en utilisant
    une conversion HSV) et affiche l'image obtenue'''
    im = Image.open(imagein)
    im = im.convert('HSV')
    for i in range(im.width) :
        for j in range(im.height) :
            valeur = im.getpixel((i, j))
            im.putpixel((i, j), (valeur[0], 0, valeur[2]))
    im.show()
```

**Exercice 4** Programmer une fonction qui prend en argument une image (couleur ou en niveaux de gris), et affiche son négatif.



(a) Image originale



(b) Niveaux de gris, méthode 1 (★)



(c) Niveaux de gris, méthode 2



(d) Niveaux de gris, méthode 3



(e) Binarisation de ★, seuil à 100



(f) Binarisation de ★, seuil à 200



Code Python :

```
def negatif(imagein) :
    '''charge l'image en couleur ou niveaux de gris contenue
    dans le fichier imagein, et affiche son négatif'''
    im = Image.open(imagein)
    for i in range(im.width) :
        for j in range(im.height) :
            val = im.getpixel((i, j))
            if im.mode == 'L' :
                im.putpixel((i, j), 255-val)
            elif im.mode == 'RGB' :
                im.putpixel(
                    (i, j),
                    (255-val[0], 255-val[1], 255-val[2])
                )
            elif im.mode == 'RGBA' :
                im.putpixel(
                    (i, j),
                    (255-val[0], 255-val[1], 255-val[2], val[3])
                )
```



(a) Négatif de l'image originale

**Exercice 5** (★) Un peu plus difficile : programmer une fonction (naïve<sup>2</sup>) de détection de contours sur une image en niveaux de gris.

Le contour sera une image en noir et blanc, avec la convention qu'un pixel est noir s'il appartient à un contour, et blanc sinon.

On considérera qu'un pixel appartient à un contour si la différence avec l'un de ses 4 voisins dépasse un seuil fixé - par exemple 50.

La fonction affichera l'image ainsi obtenue.



Code Python :

```
def voisins(i,j) :
    return [(i+1, j), (i-1, j), (i, j+1), (i, j-1)]

def contours(imagein, seuil) :
    '''charge l'image en niveaux de gris contenue
    dans le fichier imagein, et effectue une détection
    de contour avec le seuil indiqué'''
    im = Image.open(imagein)
    r = Image.new('L', (im.width, im.height), 'white')
    for i in range(1, im.width-1) :
        for j in range(1, im.height-1) :
            valeur = im.getpixel((i, j))
            for e in voisins(i,j) :
                valeur2 = im.getpixel(e)
                if abs(valeur - valeur2) > seuil :
                    r.putpixel((i, j), 0)
                    break
    r.show()
```

**Exercice 6** (★) Un peu plus délicat : écrire une fonction `augmente_contraste(imagein, imageout)`, qui prend en argument une image `imagein` en niveaux de gris, et en augmente le contraste, en appliquant une même fonction affine à chaque pixel, de sorte que les intensités de l'image résultante aillent de 0 à 255. Elle sauvegardera l'image obtenue dans le fichier `imageout`. On pourra tester sur l'image `escargotsNBBASCONTRASTE.JPG`.

2. L'extraction de contour est un problème complexe relevant de la *vision par ordinateur*, et qui fait encore l'objet de recherches actives à l'heure actuelle. Les méthodes non-triviales sont probablement hors de portée de Lycéens.



(a) Contours de \*, seuil à 50



(b) Contours de \*, seuil à 25



Code Python :

```
def trouve_affine(x1, y1, x2, y2) :
    '''renvoie le couple (a,b) tel que la droite affine
       d'équation y = ax+b passe par (x1, y1) et (x2, y2).
       Seul le cas général est traité'''
    a = (y2 - y1)/(x2 - x1)
    b = y1 - a*x1
    return (a,b)

def augmente_contraste(imagein, imageout) :
    '''augmente le contraste de l'image en niveaux
       de gris imagein en appliquant à chaque pixel
       une même fonction affine, de sorte que les valeurs
       de l'image de sortie imageout aillent de 0 à 255'''
    im = Image.open(imagein)
    #recherche des minimum et maximum d'intensité
    (m, M) = (255, 0)
    for i in range(im.width) :
        for j in range(im.height) :
            valeur = im.getpixel((i, j))
            M = max(M, valeur)
            m = min(m, valeur)
    # on trouve la fonction affine
    # qui envoie m sur 0 et M sur 255
    (a, b) = trouve_affine(m, 0, M, 255)
    # puis l'applique à chaque pixel
    r = Image.new('L', (im.width, im.height))
    for i in range(im.width) :
        for j in range(im.height) :
            nouvelle_valeur = round(a*im.getpixel((i,j))+b)
            r.putpixel((i, j), nouvelle_valeur)
    r.save(imageout)
```



(a) Image à bas contraste



(b) Image précédente avec contraste augmenté



## Chapitre 2

# Algorithmes sur une table de données

Dans toute cette section, on travaillera dans le fichier *tabledonnees.py* fourni. Les corrections des exercices sont en bas de ce fichier.

On va considérer dans cette section une base de données très simple, permettant de gérer un carnet d'adresses électroniques, et représentée sous forme de **table** (on parle également de **collection**) :

Nom	Adresse mail
John Doe	johndoe@protonmail.com
Frank Zappa	frankzappa@opensourceemail.com
Marcel Gotlib	marcel@pilote.fr
⋮	⋮

Chaque ligne à partir de la 2<sup>ème</sup> correspond à un **objet** de la collection. Chaque colonne correspond à un **attribut** (ou descripteur).

À l'intersection des lignes et des colonnes se trouvent les **données** proprement dites.

Une façon naturelle de représenter ces données en *Python* est d'utiliser une liste de tuples de chaînes de caractères.



Code Python :

```
t = [  
    ('John Doe', 'johndoe@protonmail.com'),  
    ('Frank Zappa', 'frankzappa@opensourceemail.com'),  
    ('Marcel Gotlib', 'marcel@pilote.fr'),  
]
```

Afin de tester vos fonctions, vous pouvez importer de faux carnets d'adresses à l'aide de la fonction `importer_carnet`, qui prend en argument le nom du fichier. Il y a différents fichiers disponibles, le nombre dans le nom correspondant au nombre d'objets dans le carnet :

- `carnet100.csv` (≈5ko)
- `carnet1000.csv` (≈50ko)
- `carnet10000.csv` (≈500ko)
- `carnet100000.csv` (≈5Mo)
- `carnet1000000.csv` (≈50Mo)

On pourra par exemple taper `c = importer_carnet('carnet100.csv')`<sup>1</sup>.

Ces fichiers ont été générés aléatoirement à partir d'une base de noms et de prénoms communs. Tous contiennent une entrée ('Mickael Pechaud', 'mickaelpechaud@protonmail.com'), qui vous permettra de tester vos fonctions.

1. Il faut pour cela que le *répertoire de travail* soit le répertoire contenant ce fichier. Pour voir ce répertoire, après avoir importé le module `os` (`import os`), il faut utiliser la commande `os.getcwd()` (pour **c**urrent **w**orking **d**irectory). Pour le modifier, on utilise `os.chdir('/chemin/vers/le/nouveau/repertoire')`.

## 2.1 Recherche dans une telle base

Voici une fonction permettant d'effectuer une recherche par nom dans une telle structure.



Code Python :

```
def rechercheParNom(table, nom) :
    '''renvoie une adresse mail de la personne ayant le nom passé en paramètre
    si elle apparait dans la table, et renvoie None sinon'''
    for e in table :
        if e[0] == nom :
            return e[1]
    return None
```

**Exercice 7** Lire et comprendre cette fonction, et la tester sur les carnets fournis en exemple.

### Exercice 8 – Différentes variantes d'algorithmes de recherche

1. Écrire une fonction `rechercheParAdresse(table, adresse)`, qui prend en argument une table et une adresse, et renvoie le nom de la personne dont c'est l'adresse.



Code Python :

```
def rechercheParAdresse(table, adresse) :
    '''renvoie le nom de la personne dont l'adresse mail
    est passé en paramètre si elle apparait dans la table,
    et renvoie None sinon'''
    for e in table :
        if e[1] == adresse :
            return e[0]
    return None
```

2. Écrire une fonction `nombreDAdresses(table, nom)` qui prend en argument une table et un nom, et renvoie le nombre d'adresses mail de la personne correspondante dans la table (une même personne peut avoir plusieurs adresses!).



Code Python :

```
def nombreDAdresses(table, nom) :
    '''renvoie le nombre d'adresses de la personne
    ayant le nom passé en paramètre'''
    c = 0 # c est un compteur
    for e in table :
        if e[0] == nom :
            c = c + 1
    return c
```

3. Écrire une fonction `listeAdresses(table, nom)` qui renvoie la liste de toutes les adresses de la personne. Indication : on pourra partir d'une liste vide `l=[]`, et utiliser l'instruction `l.append(adresse)` pour lui rajouter une nouvelle adresse (cette instruction ne renvoie rien, elle *modifie* la liste `l`).



Code Python :

```
def listeAdresses(table, nom) :
    '''renvoie la liste des adresses de la personne ayant
       le nom passé en paramètre'''
    l = []
    for e in table :
        if e[0] == nom :
            l.append(e[1])
    return l
```

4. Écrire une fonction `nomsParDomaine(table, domaine)` qui renvoie une liste de toutes les personnes ayant une adresse dans le domaine passé en paramètre (qui peut être par exemple `'protonmail.com'`). On pourra utiliser la commande `in` pour tester la présence d'une chaîne de caractères dans une autre :



Code Python :

```
>>> 'protonmail.com' in 'mickaelpchaud@protonmail.com'
True
>>> 'ac-montpellier.fr' in 'mickaelpchaud@protonmail.com'
False
```



Code Python :

```
def nomsParDomaine(table, domaine) :
    '''renvoie la liste des noms des personnes ayant une
       adresse dans le domaine passé en paramètre'''
    l = []
    for e in table :
        if domaine in e[1] :
            l.append(e[0])
    return l
```

## 2.2 Un problème de temps d'exécution

Exercice 9 – La recherche naïve est de complexité linéaire

1. En fonction de la longueur de la table, quelle est l'ordre de grandeur du nombre d'instructions effectuées par les différentes fonctions programmées ci-dessus ?

Il est proportionnel à la longueur  $n$  de la table. On parle de *complexité<sup>a</sup> linéaire.*, ou en  $O(n)$ .

<sup>a</sup>. Plus de précisions sur la notion de complexité en annex

2. Tester les instructions suivantes :



Code Python :

```
# importation d'une table
t = importer_carnet('carnet1000.csv')

# évaluation empirique du temps d'exécution de nomsParDomaine
c = time()
nomsParDomaine(t, 'ac-montpellier.fr')
print (time() - c)
```

Extrapoler le temps d'exécution nécessaire pour une table à  $10^5$  et  $10^6$  lignes, et tester. On peut également tester pour  $10^7$  et  $10^8$  en insérant avant la ligne `c = time()` une instruction du type `t = 10*t` ou `t = 100*t` pour dupliquer le carnet en 10 ou 100 exemplaires.

Éviter de tester avec  $10^{10}$ , sous peine de perdre la main sur la machine<sup>2</sup>.

Pour `carnet1000.csv`, un essai donne sur ma machine  $0.000232s$ .

La complexité étant linéaire, on peut penser que les temps d'exécution vont être environ 10 fois supérieurs pour `carnet10000.csv`, et 100 fois supérieurs pour `carnet100000.csv`. Des test donnent respectivement :

- $0.00217s$
- $0.0215s$

ce qui paraît cohérent.

### Exercice 10 – Recherche par Dichotomie

Dans la suite, on va travailler avec des carnets triés par ordre alphabétique des noms. Vous pourrez effectuer les tests sur carnets suivants :

- `carnet_trie100.csv`
- `carnet_trie1000.csv`
- `carnet_trie10000.csv`
- `carnet_trie100000.csv`
- `carnet_trie1000000.csv`

1. On suppose maintenant que la table est triée par ordre alphabétique des noms. On peut alors utiliser une stratégie de recherche d'une adresse mail connaissant le nom de la personne par *dichotomie* : on regarde une ligne au milieu de la table. Si l'on est tombé sur le bon nom, c'est terminé. Sinon, on

<sup>2</sup>. Représenter en machine un couple (nom, adresse) prend au moins un bit (en pratique se sera beaucoup plus, selon la façon dont les tuples et les chaînes de caractères sont stockées en mémoire) et donc représenter un carnet de longueur  $10^{10}$  prend au moins  $10^{10} \text{ bits} > 1Go$ . Cela sature la mémoire vive de la machine, qui se met à devoir écrire des données sur le disque dur, ce qui est beaucoup plus lent. On dit en bon français que la machine se met à *swapper* – ce qui est généralement à éviter...

recommence en restreignant la recherche à une moitié des lignes, selon que le nom que l'on cherche est situé avant ou après le nom trouvé par ordre alphabétique.

Afin de coder cela, la stratégie classique consiste à maintenir deux indices **d** et **f** pour **début** et **fin** et de garantir à tout moment que, si le nom recherché est présent dans la table, il est forcément situé entre les lignes **d** et **f** (au sens large).

Compléter la fonction suivante, qui utilise ce principe.



Code Python :

```
def rechercheDichotomiqueParNom(table, nom) :
    '''renvoie une adresse de la personne dont le nom est
       passé en paramètre, la table étant triée par ordre
       alphabétique des noms'''
    d = 0
    f = len(table) - 1
    m = (d+f)//2 # indice au milieu
    while d <= f :
        nommilieu = table[m][0]
        if nommilieu == nom :
            ...
        elif nommilieu < nom :
            d = ...
            f = ...
        else : # nommilieu > nom
            d = ...
            f = ...
        m = (d+f)//2
    return ...
```



Code Python :

```
def rechercheDichotomiqueParNom(table, nom) :
    '''renvoie une adresse de la personne dont le nom est
       passé en paramètre, la table étant triée par ordre
       alphabétique des noms'''
    d = 0
    f = len(table) - 1
    m = (d+f)//2 # indice au milieu
    while d <= f :
        nommilieu = table[m][0]
        if nommilieu == nom :
            return table[m][1]
        elif nommilieu < nom :
            d = m + 1
        else : # nommilieu > nom
            f = m - 1
        m = (d+f)//2
    return None
```

2. Testez empiriquement le temps d'exécution de la fonction précédente sur des exemples – et appréciez le gain obtenu. Ajouter des instructions à votre fonction pour qu'elle affiche le nombre de passages effectués dans la boucle `while`.

Quelques tests.

`carnet_trie10000.csv` : 14 passages, 0.00028s.

`carnet_trie100000.csv` : 17 passages, 0.00036s.

`carnet_trie1000000.csv` : 20 passages, 0.00040s.

3. Dans cette question, on se propose de **prouver différentes propriétés** de l'algorithme de dichotomie.

**Terminaison** : Lorsqu'un algorithme contient une boucle `while`, un problème de *terminaison* peut se poser. C'est par exemple le cas dans l'exemple suivant :

 **Code Python** :

```
n = 0
while n >= 0 :
    n = n + 1
    print(n)
```

Il est donc important de garantir que l'algorithme termine. Pour cela, une méthode classique est d'exhiber une quantité, qui tant que l'on est dans la boucle, est **entière, positive**, et qui **décroit strictement** à chaque itération de la boucle. Un petit raisonnement par l'absurde montre alors que la boucle ne peut pas se répéter indéfiniment. Exhiber une telle quantité pour prouver que l'algorithme de dichotomie termine.

$f - d$  satisfait les différentes conditions décrites.

**Complexité** : Il faut également s'interroger sur le temps d'exécution. Montrer que pour un carnet d'adresses de longueur  $2^p - 1$ , il y a au plus  $p$  passages dans la boucle. En déduire que pour un carnet de longueur  $n$ , le nombre maximal de passages dans la boucle est de l'ordre d'une constante multipliée par  $\log(n)^3$ . On parle de *complexité logarithmique*, ou  $O(\log(n))$ .

On procède par récurrence sur  $p \in \mathbb{N}^*$ .

**Initialisation** : pour  $p = 1$ , le carnet est de longueur  $2^p - 1 = 1$ . Il y a alors un passage dans la boucle.

**Hérédité** : soit  $p \in \mathbb{N}^*$ . On suppose le résultat vrai au rang  $p$ . On considère un carnet d'adresses de longueur  $2^{p+1} - 1$ . Si l'élément du milieu correspond au nom recherché, on ne fera qu'un passage dans la boucle. Sinon, on se ramène après le premier passage dans la boucle à la recherche dichotomique dans un sous-carnet d'adresses de longueur  $((2^{p+1} - 1) - 1)/2 = 2^p - 1$ . Par hypothèse de récurrence, il restera alors au plus  $p$  passages dans la boucle. Donc il y a bien au plus  $p + 1$  passages au total.

On montre par une récurrence forte sur la taille du carnet que le nombre maximal d'itérations est une fonction croissante de la taille du carnet.

Soit  $n \in \mathbb{N}^*$ . On cherche  $p \in \mathbb{N}$  tel que  $2^p - 1 \leq n < 2^{p+1} - 1$ . Cela équivaut à  $p = \lfloor \log(n + 1) \rfloor$ .

On en déduit que le nombre maximal de passages dans la boucle pour un carnet de taille  $n$  est compris entre  $\lfloor \log(n + 1) \rfloor$  et  $\lfloor \log(n + 1) \rfloor + 1$ , ce qui est bien l'ordre de grandeur demandé.

3. Où  $\log$  désigne le logarithme en base 2 – mais cela n'a pas d'importance, les différents logarithmes étant proportionnels entre eux

**Correction :** enfin, on peut prouver que l'algorithme est correct, c'est-à-dire qu'il permet bien de trouver un élément du carnet d'adresses correspondant au nom passé en paramètre s'il y apparaît. Pour cela on exhibe en général un **invariant de boucle** : il s'agit d'une propriété liant les variables du programmes, qui doit être vraie à chaque fois que l'on passe par un point donné du programme. Pour l'algorithme de dichotomie, un tel invariant pourrait être, à la ligne suivant le `while` :

« Si le nom passé en paramètre apparaît dans la table, il apparaît nécessairement entre les données d'indice `d` et `f` (dans cet ordre). »

Cette propriété est vraie avant le premier passage dans la boucle, et reste vraie d'un passage à l'autre – par construction des nouvelles valeurs de `d` et `f`.

Si la boucle se termine, c'est que `f < d`. Comme l'invariant ci-dessus est toujours vérifié, et qu'il n'y a plus de données entre les indices `d` et `f` (dans cet ordre), c'est que le nom passé en paramètre n'apparaît pas dans la table. Il est donc correct que la fonction renvoie `None`. Si le nom apparaît dans la table, cela ne peut pas se produire, et l'on est donc assuré que le `return` à l'intérieur de la boucle aura provoqué une sortie de la fonction, i.e. que le nom aura été trouvé.

Les invariants peuvent se matérialiser dans le code à l'aide de la commande `assert`. Celle-ci prend en argument une expression booléenne, ne fait rien si elle s'évalue à vrai, et provoque une erreur sinon. On peut par exemple ajouter sur la ligne suivant le `while` l'instruction suivante, correspondant à notre invariant :

```
assert(table[d][0] <= nom <= table[f][0])
```

Vous pouvez vérifier qu'un message d'erreur est renvoyé si vous essayez d'utiliser l'algorithme de recherche par dichotomie sur une table non-triée – pour laquelle il y a toutes les chances que l'invariant soit faux.

4. Sachant qu'il est possible de trier une table de  $n$  lignes en un temps au plus proportionnel à  $n \log(n)$  (on parle de *complexité quasi-linéaire*, ou  $O(\log(n))$ ), discuter la pertinence du fait de pré-trier des données pour faire des recherches dessus.

Si l'on doit chercher  $p$  données dans une table de longueur  $n$  :

- sans trier la table, cela prendra  $O(np)$  opérations.
- en triant la table une fois pour toute, cela prendra  $O(n \log(n) + p \log(n))$  opérations.

Cela devient très vite intéressant, dès que  $p$  dépasse l'ordre de grandeur de  $\log(n)$ .

5. On voudrait pouvoir effectuer des recherches rapides à la fois par nom et par adresse. Proposer une solution simple. Quel est son inconvénient ?

Une possibilité est d'avoir deux versions de la table de données :

- l'une triée par noms ;
- l'autre triée par adresses.

Deux inconvénients parmi d'autre :

- on double la mémoire nécessaire au stockage de la table ;
- on court un risque d'avoir des incohérences entre les deux copies de la table. Que faire si une faute été corrigé dans l'une des copies mais pas dans l'autre ?

Pour information, les bases de données réelles utilisent des mécanismes algorithmiques complexes appelés *index* – implémentés notamment à l'aide de structure d'arbres appelés *B-trees* – permettant, sans dupliquer les données, d'effectuer des opérations rapidement sur n'importe quelle colonne d'une table.

La duplication de données existe également, mais dans le seul but de gérer des pannes.

La façon dont sont stockées des données est donc au moins aussi primordiale que les algorithmes utilisés.

## 2.3 Recherche de doublons

### Exercice 11

1. Écrire une fonction `doublon(table)`, qui prend en argument une table **non nécessairement triée**, et teste si une même ligne y apparaît plusieurs fois (on utilisera 2 boucles imbriquées).



Code Python :

```
def doublon(table) :
    '''teste si un même nom apparaît plusieurs fois
    dans la table'''
    n = len(table)
    for i in range(n) :
        for j in range(i+1, n) :
            if table[i][0] == table[j][0]
                and table[i][1] == table[j][1] :
                return True
    return False
```

2. Évaluer l'ordre de grandeur du nombre d'opérations effectuées en fonction de la longueur  $n$  de la table. On parle de *complexité quadratique*, ou  $O(n^2)$ .

Le nombre de passages dans la double boucle est  $\sum_{i=0}^{n-1} n - i - 1$  (pour un indice  $i$  donné, on compare l'élément correspondant de la liste avec les  $n - 1 - i$  éléments suivants), dont le terme dominant est  $\frac{n^2}{2} = O(n^2)$ .

3. Tester empiriquement son temps d'exécution sur des carnets de longueur 100 ou 1000, dans le cas le pire où il n'y a pas de doublon. Extrapoler pour des valeurs plus grandes.

On obtient respectivement  $2ms$  et  $200ms$ . Au vu de la complexité, on peut s'attendre ce que le temps d'exécution soit multiplié par 100 lorsque la taille des données est multipliée par 10. On peut donc tabler sur  $20s$  pour 10000 lignes, et  $2000s \approx 30min$  pour 100000 lignes. Cela ne se produit pas sur les carnets donnés en exemple avec la fonction ci-dessus, car des doublons y apparaissent, et l'on sort prématurément des boucles imbriquées. Mais c'est un ordre de grandeur raisonnable pour le cas le pire, où il n'y a pas de doublons dans le carnet.

4. Proposer une méthode plus efficace pour effectuer ce test (on pourra commencer par trier les données!).

On commence à trier le carnet par ordre alphabétique des noms (en temps  $O(n \log(n))$ , cf ci-dessus) puis des adresses, puis on a juste à parcourir une fois le carnet et à tester ses éléments consécutifs (en temps  $O(n)$ ). Cela permet d'obtenir des temps d'exécution raisonnables pour des carnets beaucoup plus gros qu'avec la méthode précédente...

## 2.4 Trier des données

On s'intéresse dans cette section à des algorithmes de tri qui prennent en entrée un tableau, et le modifient ou renvoient un nouveau tableau contenant les mêmes éléments (avec leur multiplicité) classés par ordre croissant selon un certain critère.

Il s'agit d'une tâche algorithmique essentielle : comme nous l'avons vu avec les exemples de la dichotomie et du calcul des doublons, elle permet de *structurer les données*, ce qui ouvre la voie à des algorithmes plus rapides.

Le tri de tableaux est déjà implémenté en *Python*<sup>4</sup> :

 Code Python :

```
>>> a = [1, 2, 3, 1, 2, 1, 6, 1, 2]
>>> a.sort()
>>> a
[1, 1, 1, 1, 2, 2, 2, 3, 6]
```

Pour trier un carnet d'adresses `carnet`, on peut par exemple utiliser la commande suivante :

 Code Python :

```
| carnet.sort(key = lambda t : t[0]+t[1])
```

Détaillons son fonctionnement :

- `lambda t : t[0]+t[1]` utilise le mot clef `lambda`<sup>5</sup> et est une façon de définir une fonction sans la nommer. C'est l'équivalent mathématique de  $x \mapsto x[0] + x[1]$ .
- `key = f` signifie que la fonction  $f$  sera utilisée pour trier la liste. Plus précisément, si deux éléments `e1` et `e2` de la liste doivent être comparés, le test `f(e1) < f(e2)` sera effectué.

Dans la commande ci-dessus, on indique juste que le carnet sera trié par ordre alphabétique croissant des chaînes constituées du nom et de l'adresse accolées.

Dans la suite, on va détailler le fonctionnement de trois algorithmes de tri. Afin de ne pas alourdir, nous allons détailler le fonctionnement de ces algorithmes sur des listes d'entiers ou de réels, mais il est tout à fait possible une fois leur principe compris de les adapter aux carnets d'adresses.

Dans le fichier du TP, vous trouverez des fonctions graphiques permettant de visualiser le déroulement de ces algorithmes sous forme d'animation.

Il est essentiel de tester au fur et à mesure vos fonctions à l'aide de jeux de tests. Pour le problème du tri, on peut facilement générer des listes de réels à trier de taille arbitraire, à l'aide de commandes du type<sup>6</sup>

```
l = [random()*100 for _ in range(30)]
```

La fonction `est_triee` fournie dans le fichier (mais que vous pouvez également recoder pour l'exercice) vous permettra de déterminer si une liste est bien triée ou pas.

4. `sort` est basé sur l'algorithme *Timsort*, qui est un mélange de tri fusion et de tri par insertion.  
 5. Une notation issue du  $\lambda$ -calcul, qui est un système formel de manipulation de fonctions, sur lequel est entièrement basé le langage fonctionnel *Caml*, qui est enseigné en option informatique en MPSI.  
 6. Après avoir importé le module `random` : `import random`.

### 2.4.1 Tri par sélection

Le tri par sélection consiste à trouver le maximum du tableau, le permuter avec le dernier élément du tableau, trouver le deuxième plus grand élément du tableau, le permuter avec l'avant-dernier élément, et ainsi de suite.

1. Écrire et tester une fonction `maximum(l,i)`, qui renvoie l'indice où se trouve l'élément maximal du tableau `l` situé avant l'indice `i` (avant au sens strict).

 Code Python :

```
def maximum(l,i) :
    m = 0
    for k in range(i) :
        if l[m] < l[k] :
            m = k
    return m
```

2. Écrire une fonction `triselection(l)`, qui effectue un tri par sélection sur `l`.

 Code Python :

```
def triselection(l) :
    n = len(l)
    for k in range(n) :
        m = maximum(l, n-k)
        (l[m], l[n-1-k]) = (l[n-1-k], l[m])
        trace(l)
```

3. Testez votre fonction sur un tableau aléatoire, généré par exemple à l'aide de `l=[random()*100 for _ in range(30)]`
4. Insérer la commande `trace(l)` dans votre fonction à chaque modification de `l`, afin de visualiser sous forme d'une animation le déroulement de l'algorithme.
5. Montrer que la complexité de cet algorithme est  $O(n^2)$  (on comptera le nombre de comparaisons).

Le nombre de comparaisons effectuées lors de l'appel de `maximum(l,i)` est  $i$ .

Le nombre de comparaisons effectuées par `triselection` sur un tableau de longueur

$n$  est donc  $\sum_{i=1}^n i = \frac{n(n+1)}{2} = O(n^2)$ .

La complexité est donc quadratique.

6. Donner un invariant de boucle<sup>7</sup> qui permette de prouver que l'algorithme de tri par sélection est correct.

---

7. Cette notion est expliquée dans la partie sur l'algorithme de dichotomie.

Cette réponse peut être adaptée selon le programme que vous avez écrit.

On peut prendre comme invariant, à mettre juste après la ligne contenant la boucle `for`, « *Les  $k$  derniers éléments du tableau sont à leur place définitive* » (où  $k$  est l'indice de la boucle).

- C'est vrai au premier passage, où  $k = 0$ .
- Si c'est vrai lors d'un passage, ce sera vrai lors du passage suivant, étant donné que le maximum des  $n - k$  premiers éléments de la liste est mis en position  $n - 1 - k$ .
- C'est donc vrai en sortie de boucle, où  $k = n - 1$  et cela signifie que tous les éléments de la liste sont à leur place.

### 2.4.2 Tri par insertion

Le tri par insertion d'un tableau  $\mathbf{t}$  de longueur  $n$  se déroule en  $n$  passes. À l'issue de la  $k^{\text{ème}}$  passe, le sous-tableau à  $k$  éléments commençant au début de  $\mathbf{t}$  est trié.

Lors de la passe  $k + 1$ , on regarde  $\mathbf{t}[k+1]$ , et on l'insère à la position correcte dans le sous-tableau trié situé à sa gauche. Pour ce faire, on peut le permuter successivement avec son voisin de gauche, jusqu'à ce qu'il soit correctement positionné.

**Tableau de départ**

3	1	2	-1	6	5
---	---	---	----	---	---

**Après la passe 1** (l'élément inséré est en rouge et entre parenthèses)

(1)	3	2	-1	6	5
-----	---	---	----	---	---

**Après la passe 2** (l'élément inséré est en rouge et entre parenthèses)

1	(2)	3	-1	6	5
---	-----	---	----	---	---

**Après la passe 3** (l'élément inséré est en rouge et entre parenthèses)

(-1)	1	2	3	6	5
------	---	---	---	---	---

...

1. Implémenter, tester, et visualiser le déroulement du tri insertion.



**Code Python :**

```
def triinsertion(l) :
    n = len(l)
    for i in range(1,n) :
        k = i
        while l[k] < l[k-1] and k > 0 :
            (l[k], l[k-1]) = (l[k-1], l[k])
            k = k-1
        trace(l)
```

- Déterminer la complexité de cet algorithme.

Chaque passe effectuée au plus  $n$  opérations, et il y a au plus  $n$  passes. Donc la complexité est quadratique ( $O(n^2)$ ) – cette complexité étant atteinte pour une liste initialement triée en sens inverse.

- Donner un invariant de boucle permettant de prouver sa correction.

On peut prendre comme invariant à la ligne suivant la boucle `for`  
 « Les  $i$  premiers éléments de la liste en partant de la gauche sont triés »  
 La preuve est alors similaire à celle du tri par sélection.

### 2.4.3 Tri fusion

Plus difficile – et pas dans les projets de programmes pour l’instant (il pourrait fort bien faire son apparition en terminale). Le *tri fusion* est le plus simple des algorithmes de tris « optimaux », dans le sens que, pour trier une liste de longueur  $n$ , il utilise un nombre d’opération de l’ordre d’une constante multipliée par  $n \log(n)$  : on parle de *complexité quasi-linéaire*, ou  $O(n \log(n))$ .

Le tri fusion est un algorithme *récuratif* :

- Si le tableau n’a qu’un élément, il est déjà trié.
- Sinon, on sépare le tableau en deux parties de longueurs égales à 1 près.
- On trie récursivement les deux parties avec l’algorithme de tri fusion.
- On fusionne les deux tableaux triés en un tableau trié.

On prendra soin de tester les fonctions écrites au fur et à mesure.

- Écrire une fonction `separe(1)` qui prend en argument un tableau de longueur au moins 2, et renvoie un couple de 2 tableaux correspondant aux 2 moitiés (à 1 élément près) du tableau 1.



Code Python :

```
def separe(l) :
    '''renvoie un couple dont le premier élément
    est la première moitié de la liste passée
    en paramètre, et donc le second élément est la
    seconde moitié (avec un arrondi le cas échéant)'''
    n = len(l)
    return (l[:n//2], l[n//2:])
```

- Écrire une fonction `fusionne(l1,l2)` qui prend en argument 2 tableaux triés de même longueur (à 1 élément près), et renvoie le tableau fusionné trié correspondant, en *temps linéaire* par rapport aux tableaux de départ.



Code Python :

```
def fusionne(l1, l2) :
    '''prend en argument deux listes déjà triées,
    et renvoie la liste triée issue de la fusion
    de ces deux listes'''
    (i1, i2) = (0, 0)
    r = []
    while i1 < len(l1) or i2 < len(l2) :
        if i2 == len(l2) or (i1 < len(l1) and l1[i1] < l2[i2]) :
            r.append(l1[i1])
            i1 = i1 + 1
        else :
            r.append(l2[i2])
            i2 = i2 + 1
    return r
```

3. Écrire enfin une fonction `trifusion(l)`, qui prend en entrée un tableau  $l$  et effectue un tri fusion sur  $l$  (elle renverra un nouveau tableau en sortie).



Code Python :

```
def trifusion(l) :
    if len(l) > 1 :
        (l1, l2) = separe(l)
        return fusionne(trifusion(l1), trifusion(l2))
    return l
```

NB : on peut montrer que le tri fusion est de complexité quasi-linéaire  $O(n \log(n))$ , et que cette complexité est optimale pour un algorithme de tri général (n'utilisant que des comparaisons entre les éléments du tableau).

## 2.5 Persistance des données

Lorsque l'on exécute l'instruction `t = [('John Doe'), 'johndoe@protonmail.com'], ...`, la liste est créée dans la mémoire vive de la machine, et disparaîtra de cette mémoire lorsque la console Python sera fermée.

Pour assurer la persistance des données, elles doivent être stockées dans un fichier. Le **format CSV**<sup>8</sup> est un format ouvert très simple permettant de faire cela. Dans un fichier *CSV*, chaque ligne correspond à une ligne du tableau, et les différentes données d'une ligne sont séparées par des virgules. C'est le format utilisé pour la base de test de ce TP.

Voici par exemple le début d'un fichier *CSV* correspondant à la table donnée en exemple ci-dessus.

8. Pour *Comma-Separated Values* – « valeurs séparées par des virgules ».

John Doe, johndoe@protonmail.com  
 Frank Zappa, frankzappa@opensourcemail.com  
 Marcel Gotlib, marcel@pilote.fr

On souhaite écrire des fonctions permettant de charger une table à partir d'un tel fichier *CSV*, ou de sauvegarder une table sous forme de fichier *CSV*. Voici quelques instructions qui vont être utiles :

- `with open(fichier, 'r') as fp :`  
`for l in fp :`  
`...`

Pour lire un fichier (le 'r' est pour read). `l` prend alors pour valeurs successives les lignes du fichier, dont on peut faire quelque chose dans la boucle. Attention, `l` contient le caractère spécial `\n` qui indique une fin de ligne.

- `s.split(',')` permet de découper une chaîne de caractère selon les virgules (on pourrait remplacer ce séparateur par tout autre symbole). Le résultat renvoyé est une liste de chaînes de caractères.

**Exercice 12** Muni de ces fonctions, réécrire la fonction `importer_carnet` que vous utilisez depuis le début du TP – qui prend en argument un nom de fichier et renvoie le carnet d'adresse correspondant, sous forme de liste de tuples.



Code Python :

```
def importer_carnet(fichier) :
    '''importe le carnet dont le nom de fichier
       est passé en paramètre'''
    r = []
    with open(fichier, 'r') as fp :
        for l in fp :
            e = l.split(',')
            if len(e) == 2 :
                r.append((e[0], e[1][ :-1]))
    return r
```

On donne maintenant les instructions

- `with open(fichier, 'w') as fp :` pour ouvrir un fichier écriture (`w`write).
- `fp.write('contenu de la ligne\n')` pour écrire une ligne. Le `'\n'` est le caractère de retour à la ligne.

**Exercice 13** Écrire une fonction `sauvert_carnet(fichier,l)`, qui prend en argument un nom de fichier, et sauve le carnet `l` dans ce fichier au format *CSV*.



Code Python :

```
def sauvert_carnet(fichier, l) :
    '''sauve le carnet l dans le fichier passé en paramètre'''
    with open(fichier, 'w') as fp :
        for e in l :
            fp.write(e[0]+'','+e[1]+''\n')
```

## Chapitre 3

# Principes (très) généraux du fonctionnement d'un moteur de recherche

On travaillera dans le fichier Python `moteurrecherche.py`.

Dans cette partie, on s'intéresse rapidement au problème de l'*indexation* d'une base de textes. Il s'agit de constituer un annuaire inversé, qui permet, à partir d'un mot clé, de trouver immédiatement les textes qui se réfèrent ou correspondent à ce mot.

On peut penser à un système de classification des ouvrages d'une bibliothèque par thèmes : l'indexation peut être faite par un humain dans ce cas, qui à chaque livre associe quelques mots clefs, puis crée ensuite l'annuaire inversé permettant de trouver l'ensemble des livres correspondant à tel ou tel mot clef.

Les moteurs de recherche comme *DuckDuckGo*<sup>1</sup> fonctionnent également à l'aide d'algorithmes d'indexation, utilisant le même principe :

- des algorithmes (appelés *robots d'indexation* (*web crawler*) en anglais), parcourent automatiquement le web pour aspirer tout ou partie de son contenu (un ordre de grandeur : en 2008, une équipe de recherche de *Google* a annoncé que le web contenait plus de  $10^{12}$  pages web) ;
- le contenu est analysé et indexé – c'est l'étape de création de l'annuaire inversé ; (éventuellement, une partie peut-être faite par les robots, pour éviter d'aspirer des données inutiles)
- lorsqu'un utilisateur du moteur de recherche tape un mot clé, il n'y a plus qu'à utiliser l'annuaire inversé pour retrouver toutes les pages pertinentes.

### 3.1 Indexation

Nous allons dans cette section décrire un algorithme d'indexation naïf.

On suppose donné un corpus de textes sous forme d'une liste de chaînes de caractères.

```
corpus = [  
'Ceci est le début du premier texte ..... ',  
'Et cela le début du second .....',  
. ,  
. ,  
. ,  
]
```

On rappelle les instructions suivantes, qu'il est recommandé de tester :

---

1. Pour citer un exemple de moteur de recherche qui respecte la vie privée de ses utilisateurs et ne les traque pas sur le web.

- `m in s` teste si la chaîne de caractères `m` apparaît comme sous-chaîne dans `s`.<sup>2</sup>
- `l.append(5)` permet d'ajouter l'élément `5` à une liste `l` (cette instruction ne renvoie rien, elle *modifie* la liste `l`).

**Exercice 14** Écrire une fonction `indexe(mot, corpus)`, qui prend en argument un mot (de type « chaîne de caractères ») et un corpus au format décrit ci-dessus, qui renvoie la liste des indices des textes du corpus où apparaît le mot. On numérottera à partir de 0, et on ne se souciera pas de mots qui peuvent-être un sous-mot d'autres. On pourra utiliser la commande `s.lower()`, qui convertit une chaîne de caractères en minuscules, pour éviter les problèmes liés aux majuscules<sup>3</sup>.

 Code Python :

```
def indexe(mot, corpus) :
    '''renvoie la liste des indices des textes du corpus
       où apparaît le mot'''
    r = []
    for i in range(len(corpus)) :
        if mot in corpus[i].lower() :
            r.append(i)
    return r
```

**Exercice 15** Tester votre fonction. Un corpus est fourni – vous pouvez le charger à l'aide de la commande<sup>4</sup> `corpus = charge_corpus('corpus.txt')`. Le corpus est constitué de cinquante textes qui sont des extraits de wikipédia (≈16000 mots, 100ko). On pourra tester la fonction avec les mots « table », « univers » ...

 Code Python :

```
>>> corpus = charge_corpus('corpus.txt')
>>> indexe('table', corpus)
[8, 13, 19, 34, 35]
>>> indexe('univers', corpus)
[2, 7, 14, 16, 19, 28, 37, 40]
```

On souhaite maintenant indexer plusieurs mots. On peut pour cela utiliser une structure *Python* appelée *dictionnaire*.

Il s'agit d'une structure de données proche d'une liste, mais où les éléments ne sont pas indicés par un entier, mais par une *clé* – qui est souvent une chaîne de caractères.

2. Il est possible de recoder cette fonction – c'est un exercice classique sur les chaînes de caractères. La version naïve va utiliser un nombre d'instructions au pire proportionnel à  $|m||s|$  (où  $|m|$  désigne la longueur de  $m$ ) – i.e.  $O(|m||s|)$ . Il existe des algorithmes plus complexes, basés sur des *automates finis*, permettant d'obtenir une complexité  $O(|s|)$  une fois une phase de « préparation » du motif effectuée.

3. Sinon, si l'on cherche le mot `bonjour` dans le texte `Bonjour monsieur l'inspecteur`, il ne sera pas trouvé.

4. Il faut pour cela que le *répertoire de travail* soit le répertoire contenant ce fichier. Pour voir ce répertoire, après avoir importé le module `os` (`import os`), il faut utiliser la commande `os.getcwd()` (pour `current working directory`). Pour le modifier, on utilise `os.chdir('/chemin/vers/le/nouveau/repertoire')`.

L'exemple suivant montre comment créer un dictionnaire, et comment accéder à des valeurs en lecture ou en écriture.



Code Python :

```
>>> d = {'clé1' : 'valeur1', 'clé2' : 'valeur2', 'clé3' : 'valeur3'}
>>> d['clé1']
'valeur1'
>>> d['clé4']
KeyError : 'clé4'
>>> d['clé2'] = 'nouvellevaleur'
>>> d
{'clé3' : 'valeur3', 'clé2' : 'nouvellevaleur', 'clé1' : 'valeur1'}
>>> del d['clé2'] #suppression d'une clé
>>> d
{'clé3' : 'valeur3', 'clé1' : 'valeur1'}
```

Et pour parcourir un dictionnaire d :



Code Python :

```
for k in d.keys() : #liste des clés
    print ('clé : ', k, ' -- valeur : ', d[k])
```

Finalement, on peut créer un dictionnaire vide à l'aide de la commande



Code Python :

```
d = {}
```

**Exercice 16** Écrire une fonction `annuaire(listemots, corpus)`, qui prend en argument une liste de mots, un corpus, et renvoie le dictionnaire correspondant.

Testez par exemple avec `"listemots = ['univers', 'mesure', 'suède', 'européenne']"`.



Code Python :

```
def annuaire(listemots, corpus) :
    '''renvoie un dictionnaire correspondant à
       l'indexation des mots de la liste dans le corpus'''
    d = {}
    for mot in listemots :
        d[mot] = indexe(mot, corpus)
    return d

>>> dico = annuaire(
    ['univers', 'mesure', 'suède', 'européenne'],
    corpus)
>>> dico
{'univers' : [2, 7, 14, 16, 19, 28, 37, 40],
 'mesure' : [11, 27, 28, 29, 31, 32, 40],
 'suède' : [31, 32, 40],
 'européenne' : [16, 18, 33, 39, 40, 43]}
>>> dico('mesure')
[11, 27, 28, 29, 31, 32, 40]
```

Pour rechercher un mot, il suffit maintenant d'utiliser la fonction suivante, où `dico` est le dictionnaire calculé par la fonction précédente, et qui renvoie rapidement<sup>5</sup> la liste des indices des textes où apparaît le mot demandé.



Code Python :

```
def recherche(mot, dico) :
    return dico[mot]
```

Pour finir, vous trouverez dans le fichier *Python* une fonction `tous_mots` qui, appelée sur le corpus, permet de récupérer la liste de tous les mots d'au moins 5 lettres y apparaissant. On peut alors exécuter l'instruction suivante :



Code Python :

```
d = annuaire(tous_mots(c), c)
```

qui crée une indexation du corpus sur tous ces mots en quelques secondes.

En parcourant le dictionnaire obtenu, vous pouvez voir que subsistent de nombreux problèmes, plus ou moins simples à résoudre :

- un mot et son pluriel sont différenciés ;
- il reste des apostrophes (mais dans certains cas, on peut vouloir les garder – dans « aujourd'hui », dans certains noms propres...);

5. Les dictionnaires *Python* sont implémentés à l'aide d'une structure appelée *table de hachage*, qui garantit des accès par clé aux éléments du dictionnaire en temps moyen constant – en lecture et en écriture.

- certains mots – comme « pendant », ou « celle-ci » – ne vont pas être très utiles pour faire des recherches.

**Exercice 17** Pouvez-vous évaluer la complexité de la construction du dictionnaire en fonction de la taille du corpus  $n$  (en nombre de caractères) et du nombre de mots  $p$  à indexer ?

Une question pas évidente, qui dépend de la complexité de fonctions utilisées dans les fonctions que nous avons écrites.

L'indexation d'un mot nécessite a minima de parcourir une fois le corpus (on peut effectivement s'en sortir avec un seul parcours, cf note ci-dessus).

L'indexation de l'ensemble des mots sera donc de complexité  $O(np)$ , en admettant que les opérations sur un dictionnaire se font en temps constant.

Vue la taille du web, avoir des algorithmes d'indexation très rapides est essentiel. Il existe par exemple des algorithmes d'indexation – basés sur des structures d'arbres – permettant d'indexer un ensemble prédéterminé de mots sur un corpus en parcourant une seule fois ce corpus.

L'indexation proposée ici est purement syntaxique – il peut également être intéressant pour des moteurs de recherche d'avoir des informations d'ordre sémantique, c'est-à-dire prenant en compte d'une façon ou d'une autre le sens des pages web.

Afin d'aider à l'indexation, on peut lorsque l'on écrit une page web, y spécifier des mots clefs pertinents.

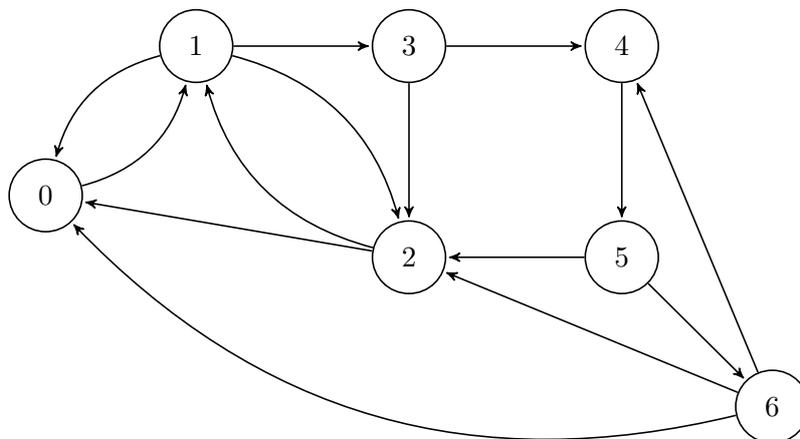
## 3.2 Calcul de popularité sur un exemple jouet

Un autre problème est le choix d'un ordre dans lequel afficher les résultats d'une recherche<sup>6</sup>.

L'idée de base est de voir le web comme un graphe dont les sommets sont les pages et les arcs les liens entre les pages. On parcourt aléatoirement ce graphe (ce qui revient à faire comme si une personne naviguant sur le web cliquait au hasard sur des liens), et l'on peut mesurer la popularité d'une page à l'aide de la probabilité de se retrouver sur celle-ci au bout d'un certain temps (en particulier, plus il y a de liens pointant vers cette page, plus elle sera populaire, en particulier si ces liens sont dans des pages populaires...).

La théorie sous-jacente est celle des chaînes de Markov<sup>7</sup>.

Afin de comprendre cela, on peut réaliser de petites simulations sur des exemples jouets. On va dans la suite considérer le graphe suivant :



6. Dans lequel le problème de la neutralité du moteur de recherche peut se poser.

7. La distribution de probabilité de se retrouver dans telle ou telle page au bout d'un temps long correspond à la distribution stationnaire de cette chaîne, qui s'obtient comme vecteur propre d'une certaine matrice – appelée *matrice de transition* du graphe. Cf par exemple la page wikipédia des chaînes de Markov. C'est l'idée de base des algorithmes de type *Page Rank*. Pour le web entier, il s'agit d'une matrice carrée de côté  $\sim 10^{12}$ , et on a donc recours à des algorithmes d'approximation. Les méthodes réellement utilisées ne sont généralement pas publiques.

Chaque sommet correspond à une page web, et un arc entre 2 sommets signifie qu'il y a, dans la page web à l'origine de l'arc, un lien vers la page web correspondant à l'autre extrémité de cet arc.

En machine, un tel graphe pourra être représenté par sa *liste d'adjacence*. Il s'agit d'une liste dont le  $i^{\text{ème}}$  élément est la liste des voisins sortants du sommet  $i$ <sup>8</sup>. Par exemple, pour le graphe ci-dessus, on obtient :

```
g = [
[1], #voisins sortants de 0
[0,2,3], #voisins sortants de 1
[0,1], #voisins sortants de 2
[2,4], #voisins sortants de 3
[5], #voisins sortants de 4
[2,6], #voisins sortants de 5
[0,2,4], #voisins sortants de 6
]
```

On donne pour l'exercice suivant la commande `randint(a,b)`, du module `random`, qui renvoie un entier aléatoire choisi uniformément dans  $[[a, b]]$ .

**Exercice 18** Écrire une fonction `marchealeatoire(g, s, l)` qui prend en argument un graphe, un numéro de sommet, et une longueur  $l \in \mathbb{N}^*$  et effectue une marche aléatoire de longueur  $l$  à partir de ce sommet (à chaque étape on choisit aléatoirement de façon uniforme un voisin sortant du sommet courant parmi les sommets possibles.). Elle renverra le numéro du sommet atteint.



Code Python :

```
def marchealeatoire(g, s, l) :
    '''effectue une marche aléatoire de longueur l
    en partant du sommet s dans le graphe g - représenté
    par une liste d'adjacence. Renvoie le numéro du
    sommet atteint.'''
    for _ in range(l) :
        s = g[s][randint(0, len(g[s])-1)]
    return s
```

**Exercice 19** Écrire une fonction `popularite(g, l, n)` qui prend en argument un graphe, une longueur  $l \in \mathbb{N}^*$  et un entier  $n \in \mathbb{N}^*$ , et effectue  $n$  marches aléatoires de longueur  $l$  sur le graphe. Elle renverra une liste dont le  $i^{\text{ème}}$  élément est le nombre de fois où le sommet  $i$  du graphe a été atteint par une marche aléatoire.

8. Cela correspond donc à la liste des pages web accessibles en 1 click à partir de la page.



Code Python :

```
def popularite(g, l, n) :
    '''renvoie une mesure de popularité du graphe g - en
    effectuant n marches aléatoires de longueur l
    sur le graphe'''
    r = len(g)*[0]
    for _ in range(n) :
        s = randint(0, len(g)-1)
        s2 = marchealeatoire(g, s, l)
        r[s2] += 1
    return r
```

**Exercice 20** Écrire une fonction `popularite_normalisee(g, l, n)` qui renvoie la même liste, mais normalisée de façon à ce que la somme de ses éléments vaille 1.



Code Python :

```
def popularite_normalisee(g, l, n) :
    '''renvoie une mesure normalisée de popularité
    du graphe g - en effectuant n marches aléatoires
    de longueur l sur le graphe'''
    p = popularite(g, l, n)
    return [e/n for e in p]
```

Une simulation de 1 000 000 marches aléatoires donne par exemple

```
[0.21722, 0.31733, 0.20122, 0.10587,
 0.06334, 0.06331, 0.03169]
```

à comparer au résultat théorique<sup>9</sup>

```
[41/189, 20/63, 38/189, 20/189, 4/63, 4/63, 2/63]
```

i.e. environ

```
[0.21793, 0.31746, 0.20106, 0.10582,
 0.06349, 0.06349, 0.03175]
```

9. Obtenu à l'aide du logiciel de calcul formel *Sage* en calculant le vecteur propre à gauche correspondant à la valeur propre

1 de la matrice de transition du graphe :

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1/3 & 0 & 1/3 & 0 & 0 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & 0 & 1/2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1/2 & 0 & 0 & 0 & 1/2 \\ 1/3 & 0 & 1/3 & 0 & 1/3 & 0 & 0 \end{pmatrix} . \text{ Cf note ci-dessus et un cours d'introduction}$$

aux chaînes de Markov!



# Annexe A

## Complexité

Il est question dans ce poly de *complexité* de telle ou telle fonction, ou de tel ou tel algorithme.

Sans rentrer dans les détails<sup>1</sup>, la complexité d'un algorithme est une mesure de l'ordre de grandeur du nombre d'« opérations élémentaires » (selon les cas : affectation, addition, lecture d'un élément...) nécessaires à son exécution, en fonction de la « taille des données » en entrée.

Dans ce poly, l'entrée est généralement une liste, et la « taille des données » est simplement dans ce cas sa longueur  $n$ .

La complexité est généralement donnée sous forme d'un  $O()$ <sup>2</sup>.

Voici les principales *classes de complexité* que nous aurons à manipuler.

complexité	notation	remarque
Constante	$O(1)$	nombre d'opérations indépendant de la taille du tableau
Logarithmique	$O(\log(n))$ <sup>3</sup>	
Linéaire	$O(n)$	nombre d'opérations au plus proportionnel à la taille du tableau
Quasi-linéaire	$O(n \log(n))$	log est « quasiment » constante en pratique !
Quadratique	$O(n^2)$	
Cubique	$O(n^3)$	on commence souvent à avoir des ennuis en pratique ici...
Exponentielle	$O(a^n)$ , où $a > 1$	et ici de gros ennuis...

### Quelques ordres de grandeur

Les ordres de grandeur (très généreux) ci-dessous sont donnés en supposant que la constante devant le nombre d'opérations est 1, et que l'on travaille sur un ordinateur capable d'effectuer  $10^9$  opérations par  $s$ .

Complexité	$n = 10^4$	$n = 10^6$	$n = 10^8$
$O(1)$	1 <i>ns</i>	1 <i>ns</i>	1 <i>ns</i>
$O(\log n)$	9 <i>ns</i>	14 <i>ns</i>	18 <i>ns</i>
$O(n)$	10 $\mu s$	1 <i>ms</i>	100 <i>ms</i>
$O(n^2)$	100 <i>ms</i>	15 <i>min</i>	100 jours
$O(n^3)$	15 <i>min</i>	30 <i>ans</i>	31 millions d'années

Complexité	$n = 10$	$n = 50$	$n = 100$
$O(2^n)$	1 $\mu s$	13 jours	40 000 milliards d'années

1. passionnants, mais... difficiles. Il faut se donner un *modèle de calcul* formel pour faire cela – c'est par exemple l'objet de l'étude des *Machines de Turing*.

2.  $f(n) = O(g(n))$  si et seulement si il existe une constante  $K$  telle que pour  $n$  assez grand on ait  $f(n) \leq K g(n)$ .

3. Peu importe la base du logarithme utilisée ici, mais en informatique, log désigne généralement  $\log_2$ .