Algorithmes classiques

1 Avec un tableau

Exercice 1 : Algorithmes simples avec parcours de tableau

1. Écrire *de trois façons différentes* des fonctions `tableau_aleatoire(n)` retournant une *liste* de taille `n` contenant des valeurs aléatoires entre 0 et 10.

   On pourra utiliser `randint` du module random

2. Écrire une *procédure* `affiche` affichant une liste à une dimension.

   ```
   >>> affiche([2,3,4,6])
   2 3 4 6
   >>>
   ```

3. Écrire une *procédure* `double_les_impairs(tab)` prenant une liste en paramètres, et doublant toutes les valeurs impaires de la liste.

   ```
   >>> l=[2,3,5,6]
   [2,3,5,6]
   >>> double_les_impairs(l)
   >>> l
   [2,6,10,6]
   ```

4. Écrire *de deux façons différentes* une fonction `somme(tab)` retournant la somme des éléments d'un tableau.

   On pourra faire un parcours du tableau par les indices :

   `for i in range(len(tab))`

   Ou par les valeurs :

   `for x in tab`

5. Écrire une fonction `minimum(tab)` retournant le minimum du tableau `tab`.

   ```
   >>> minimum([5,3,12,2,7])
   2
   ```

6. Modifier la fonction précédente pour retourner *l'indice du minimum* (le numéro de la case) du tableau `tab`.

   ```
   >>> indice_minimum([5,3,12,2,7])
   3
   ```

7. Écrire une fonction `indice(tab,x)` retournant la première occurrence de la valeur `x` dans le tableau `tab`, si elle existe. Et `-1` si cette valeur `x` n'est pas présente dans `tab`.

8. Écrire une fonction `triage(tab,x)` prenant un tableau `tab` et une valeur entière `x` et retournant deux listes : d'un côté, tous les éléments $< x$, de l'autre, tous les éléments $\geq x$ de `tab`.

   cette fonction peut être utilisée à la base du tri rapide.

   Remarque : dans cette fonction, il est beaucoup plus pratique d'utiliser `append` ; plus compliqué avec des tableaux de taille fixe.

2 Chaînes de caractère

Si c est une chaîne de caractères, on pourra utiliser `c[n:m]` qui retourne une copie de la sous-chaîne de `c` des indices `n` à `m-1` compris.

Exercice 2 : Echauffement sur les chaînes

Les expressions suivantes sont-elels correctes ? Quel est le résultat ?

```
int("23")               str(4)                          int(str(43))
"J'ai "+3+" ans"        "j'ai "+"trois"+" ans"          "j'ai "+str(3)+" ans"
"J'ai "+str(3+5)+"ans"          "j'ai "+str(3)+str(5)+" ans"
"Bonjour"*2                     "Bonjour"*"Bonjour"
str(int("3"+"4"))               int(str(3+4))
ord("a")                        chr("65")
chr(ord("f") - ord("a") + ord("A"))
```

Exercice 3 : Recherche de motif

Ecrire une fonction `recherche(m,t)` prenant deux chaînes de caractères `m` et `t` et recherchant la première occurence du motif `m` dans la chaîne `t`. Cette fonction retournera l'indice du début de l'occurence (et -1 si le motif `m` n'apparaît pas dans `t`).

```
>>>recherche("bra", "abracadabra")
  1
```

Ce problème a été abondamment étudié, et des solutions très efficaces (sans aucun retour en arrière dans la lecture de `t`) existent ; on attend ici un algorithme simple, pas forcément le plus efficace.

3 Chiffrement de césar et autres

Le but dans cette partie est de réaliser des petites fonctions de chiffrement autour du chiffrement de César.

Exercice 4 :

1. Écrire une fonction `maj` qui prend un texte en paramètres (supposé en `acsii`, sans accents), et passe toutes les lettres en majuscule. On supprimera tout ce qui n'est pas lettre. On retournera la chaîne obtenue.
2. Ecrire une fonction `cesar(texte, dec)` prenant un texte `texte` en paramètres, le pré-trétant avec `maj`, et lui appliquant le chiffrement de césar : chaque lettre est décalée de `dec` dans l'alphabet.

   ```
   >>> cesar("BONJOURZORG ",2)
   "DQPLQWBQTI"
   ```

   Remarque : Pour obtenir une sortie sous forme de chaîne de caractères : il n'est pas efficace de concaténer les caractères à la suite à une sortie :

```
  sortie =""

  #ensuite, pour chaque caractere a ajouter a la sortie :
  for ... in ... :
    sortie = sortie+ c
```

Chaque instruction `sortie = sortie+c` fait alors la copie de toute la liste. Il est alors plus efficace de créer la sortie sous forme de liste de caractères, puis de la convertir avec la fonction `join`

```
>>> sortie
 ['a','b','c','d','e']

>>> "".join(sortie)
 'abcde'
```

3. Écrire une fonction `decrypter_cesar(texte)`, qui tente de décrypter un texte codé par chiffrement de césar, sans connaître la clé : on essaiera de faire en sorte de faire correspondre la lettre la plus fréquente du texte avec le "E".
4. *Décryptage plus robuste* : la lettre la plus fréquente n'est pas forcément le E, en particulier dans un texte court. Pour avoir un résultat plus garanti, on peut tenir compte des statistiques de fréquence de chaque lettre.
   (a) Ecrire une fonction `statistiques(texte)` qui prend en entrée un texte supposé en majuscules et qui retourne un tableau de 26 lettres donnant les fréquences de chaque lettre dans l'ordre de l'alphabet.

```
>>> statistiques('ABRACADABRA')
[0.4545, 0.1818, 0.0909, 0.0909, 0.0, 0.0, 0.0, 0.0, 0.0,
   0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.1818, 0.0,
   0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

   (b) Ecrire une fonction `decrypter_cesar2(texte)` qui tente de décrypter le texte supposé chiffré par la méthode de César, en trouvant le décalage qui minimise la somme des différences de fréquences entre le texte et les fréquences des lettres en français.
   On pourra utiliser "en dur" un tableau des fréquences des lettres en français, calculé avec `statistiques` sur un gros texte.

   Cette seconde fonction pourra servir de base à des décryptages plus compliqués, comme le chiffrement de Vigenère, qui font intervenir des décalages.
5. Lecture et écriture dans un fichier : écrire une fonction `cesar_fichier(entree, sortie,dec)` ouvrant un fichier texte `entree`, appliquant le chiffrement de césar à ce fichier et enregistrant le résultat dans le fichier `sortie`
   On pourra utiliser :

```
fEntree = open("monFichier.txt","r")
texte = fEntree.read()
fEntree.close()
```

```
fSortie = open("monFichierSortie.txt","w")
fSortie.write(texte)
fSortie.close()

#ou bien :

with open("monFichier.txt","r") as fEntree :
    #syntaxe qui évite d'oublier de fermer le fichier.
    for line in fEntree :
        #exemple de lecture une à une des lignes du fichier
        print(line)
```

6. Ecrire une fonction inverse `dechiffrer_cesar_fichier`. Cette fonction pourra appeler `cesar_fichier` avec un décalage bien choisi.

Exercice 5 : Vigenère

Le *chiffement de Vigénère* s'appuie sur un mot-clé. Chaque lettre du mot-clé indique un décalage différent : "A" pour 0, "B" pour 1...

On répète le mot-clé au-dessus du texte clair ; puis on décale chaque lettre du texte clair du décalage donné par la lettre du mot-clé.

Exemp

mot-clé	C	O	Q	C	O	Q	C	O	Q
décalage	2	14	16	2	14	16	2	14	16
texte clair	B	O	N	J	O	U	R	Z	O
texte chiffré	D	C	D	L	C	K	T	N	E

1. Ecrire une fonction `chiffrement_vigenere(texte,mot_cle)` chiffrant le texte à l'aide du mot-clé donné en suivant le chiffrement de Vigénère.
2. Ecrire une fonction de chiffrement de Vigénère lisant et écrivant dans deux fichiers textes dont les noms sont donnés en paramètre.
3. Ecrire une fonction inverse de déchiffrement de vigénère, à mot-clé donné.
4. Écrire une fonction de décryptement de Vigénère, tentant de déchiffrer un texte chiffré par Vigénère, sans connaître le mot clé, mais en supposant connue la longueur du mot-clé.

   L'idée est que si on connaît la longueur n du mot-clé, on sait que toutes les n lettres, les lettres ont eu le même décalage de César. Il est alors possible de décrypter ce décalage de César en supposant que la lettre la plus fréquente doit être le "E".

4 Algorithmes suppélementaires sur les listes

Exercice 6 : La suite de Conway

La suite de Conway est une suite de séquences de 0 et de 1 : 0, 10, 1110, 3110, 132110, 1113122110, ...

Elle s'obtient en partant du terme précédent, et en "lisant" le nombre précédent. Par exemple, on passe de "1110" à "3110" car 1110 est constitué de :

— trois 1 : donne 31

— un 0 : donne 10

1. Écrire une fonction `conway` prenant en entrée une liste l de chiffres et calculant le terme suivant de la suite de Conway.

   Elle pourra fonctionner récursivement, en traitant la première séquence de nombres égaux au début de l, et en se réappelant récursivement sur la suite de l.

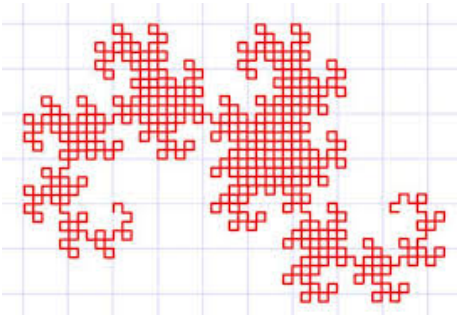
```
>>> conway([1,1,2])
[2,1,1,2]
```

2. Écrire une fonction `conway_itere` prenant en entrée une liste l et un entier n, et calculant n termes consécutifs de la suite de conway sur l.

   On procédera encore de manière récursive pour cette fonction.

Exercice 7 : Courbe du dragon

La *courbe du dragon* tire son nom de sa ressemblance avec les flammes d'un dragon. Elle peut être construite par une suite d'instructions : virage à gauche ou à droite, avancée d'un pas. Pour coder cette courbe, on construira une liste de booléens, True pour virage à gauche, False pour virage à droite.



La courbe d'ordre n est définie récursivement par

— la courbe d'ordre 1 est False.

— la courbe d'ordre n est obtenue en concaténant la courbe d'ordre $n-1$, puis False, puis la courbe d'ordre $n-1$ *à l'envers et en inversant True et False.*

Ainsi,

— `dragon(1) = [False]`.

— `dragon(2) = [False, False, True]`.

— `dragon(3) = [False, False, True, False, False, True, True]` ...

1. Ecrire la fonction `dragon` prenant un entier n en paramètre, et retournant la liste codant la courbe d'ordre n.

2. On suppose avoir importé le module Turtle avec la commande `form turtle import` .

   Ecrire une fonction `afficheDragon` prenant en paramètre un entier n, et affichant la courbe du dragon d'ordre n. On pourra utiliser les commandes

   — `left(90)` et `right(90)` pour tourner

   — `forward(50)` pour avancer.