

Exercice 1. *Codage basique.*

1. Écrire une fonction `code_une(1t)` qui, étant donnée une lettre `1t` (majuscule, sans accents), renvoie son code entre 0 et 25 attribué de la manière suivante : "A" correspond à 0, "B" correspond à 1, ..., "Z" correspond à 25.
Tester sur des exemples simples. En utilisant une boucle (cf. le cours), afficher les codes des lettres du mot "CESAR".
2. Écrire une fonction réciproque `decode_une(c)` qui étant donné un nombre (code `c`) entre 0 et 25 renvoie la lettre correspondante.
Tester.
3. Écrire une fonction `code_mot(m)` qui étant un mot (une chaîne de caractères) `m` renvoie une *liste* de codes. On pourra utiliser la commande `map` (cf. l'aide-mémoire).
Écrire une fonction `decode_mot(L)` qui étant donnée une liste de codes, renvoie un *mot* correspondant (c'est-à-dire une chaîne de caractères et non une liste de lettres).
Indication : pour transformer une liste de lettres `L` en un mot, on pourra utiliser la commande `"".join(L)` (double guillemet représente la chaîne vide).
Tester sur quelques mots si les deux dernières fonctions sont bien réciproques.

Exercice 2. *Chiffrement de César*

Le chiffrement de César consiste à remplacer chaque lettre dans une phrase par la lettre décalée d'un nombre de places dans l'alphabet. Par exemple, avec un décalage $d = 3$, "A" devient "D", "B" devient "E", etc. Ce décalage est circulaire, c'est-à-dire "X" devient "A", "Y" devient "B" et "Z" devient "C".

1. Écrire une procédure `cesar(ph, dec)` qui effectue le cryptage de César d'une phrase `ph` avec un décalage `dec`.
Indication : pour toutes les caractères de la phrase en initiale `ph`,
 - si c'est un espace, on rattache un espace à la phrase cryptée (qui se construit au fur et à mesure);
 - si c'est une lettre et non un espace, on la crypte selon l'algorithme et rattache à la phrase cryptée; pour réaliser un décalage cyclique, pensez à l'opérateur `%` ("modulo").
2. Écrire une procédure de décryptage `decesar(m, dec)`.
3. Crypter la phrase "CE MESSAGE EST CONFIDENTIEL" avec un décalage de 10.
Décrypter le résultat.
4. Sachant qu'il s'agit d'un cryptage de César, décrypter la phrase suivante (Edsger W. Dijkstra) :
"KYV RIK FW GIFXIRDDZEX ZJ KYV RIK FW FIXREZQZEX TFDGCVZKP"

Exercice 3. *Chiffrement affine*

La proposition suivante permet de construire le chiffrement dit affine.

Posons $\mathcal{A} = \{0, 1, \dots, 25\}$. Soient a et b deux entiers dans \mathcal{A} . Supposons que a est premier avec 26. Alors l'application $f : \mathcal{A} \mapsto \mathcal{A}$ définie par $f(n) = an + b \pmod{26}$ est une bijection.

1. Afficher la liste des entiers dans \mathcal{A} premiers avec 26.

2. Écrire une fonction `affine(mot,a,b)` qui réalise le cryptage affine $f(n) = an + b \bmod 26$. La fonction retournera un mot chiffré et non une liste de codes.
3. Écrire une fonction `inverse(a)` qui étant donné $a \in \mathcal{A}$ trouve, par la recherche exhaustive, son inverse modulo 26 : il s'agit d'une valeur $a' \in \mathcal{A}$ telle que $a'a = 1 \bmod 26$.
Attention : pour certaines valeurs de a , l'inverse n'existe pas; dans ce cas, la fonction pourra renvoyer 0.
Afficher les valeurs $a \in \mathcal{A}$ pour lesquelles la fonction ne trouve pas l'inverse a' . On affichera le message sous la forme suivante : *xx n'est pas inversible modulo 26*.
4. Écrire une fonction de déchiffrement affine `de_affine(mot, a, b)`.
5. Crypter le mot "CONFIDENTIEL" avec $f(n) = 5n + 7$
Décrypter. *Facultatif* : Écrire une fonction qui crypte (décrypte) une phrase entière.
6. Décoder le mot "KYBIX" étant donné qu'il a été codé avec un codage affine avec $a < 10$ et $b < 10$.
Utiliser une contrainte concernant une de ces variables pour limiter la recherche exhaustive.
7. Etant donné $a = 15$, décoder "LP NVP UJVR YCJAVXRJUR L PRG AP QH LJFYCPIPURXJU".¹

Exercice 4. Suite récurrente, fonction récursive

En mathématiques, une suite récurrente est une suite où le terme suivant est définie à l'aide d'un (ou plusieurs) termes précédents, comme par exemple la suite de Fibonacci.

Par analogie, en algorithmique, on parle d'une *fonction récursive*, c'est-à-dire une fonction qui fait appel à elle-même. Pour éviter de tourner en rond, une telle fonction possède toujours un argument entier qui décroît à chaque appel, jusqu'à ce que sa valeur descende à 0 (ou un autre entier *initial*).

En particulier, il est facile de calculer une suite récurrente à l'aide d'une fonction récursive : le code correspond naturellement à la formulation mathématique.

Soit $u_0 = 1$ et $u_n = u_{n-1} + 1/n!$, $n \geq 1$.

Écrire une fonction récursive `euler(n)` qui calcule u_n .

Calculer `euler(20)`.

Exercice 5. Limitations de récursivité

1. *Nombre d'appels en Python*

Soit $u_0 = 0$ et $u_n = u_{n-1} + 1/n$, $n \geq 1$.

Écrire une fonction récursive `harmonique(n)` qui calcule u_n .

Calculer u_{100} puis u_{950} . Peut-on calculer u_{1010} ?

¹William Thurston, médaille Fields (à propos de la recherche en mathématiques).

2. Branchements multiples

- (a) Écrire une fonction `fibo_r` qui calcule la suite de Fibonacci de manière récursive. Tester.

Mesurer le temps d'exécution de `fibo_r(25)` en utilisant le code suivant

```
from time import time
t_start = time()
fibo_r(25)
t_fin = time()
print("temps d'execution", t_fin-t_start, "sec.")
```

- (b) Calculer et afficher le temps d'exécution de `fibo_r(n)` pour n variant de 25 à 35.

Calculer et afficher aussi le rapport entre deux temps d'exécution pour deux valeurs de n consécutives. On pourra arrondir les résultats à 2 chiffres après la virgule. Que peut-on constater ? On mettra la réponse en commentaire dans le fichier.

- (c) *Estimer* le temps qui serait nécessaire pour calculer `fibo_r(80)`. Et `fibo_r(120)` ? On mettra la réponse en commentaire.

Attention ! On ne demande pas de calculer `fibo_r(80)` ! D'une manière générale, ne lancez pas un calcul de `fibo_r(n)` pour une valeur de n supérieur à 40 (risque de saturer votre machine).

Exercice 6. Fractales auto-similaires

Voici une fonction Python `triangle(x, y, c)` qui permet de dessiner un triangle équilatéral de côté c et dont le sommet en bas à gauche se situe en (x, y) :

```
import numpy as np
import matplotlib.pyplot as plt
def triangle(x, y, c):
    plt.fill([x, x+c, x+c/2], [y, y, y+c*np.sqrt(3)/2], "b")
```

1. Recopiez cette fonction dans votre fichier. Dessinez trois triangles empilés comme dans la Fig. 1, le sommet le plus haut se situant en $(0.5, \sqrt{3}/2)$.

Remarque technique : le graphique peut se cacher derrière la fenêtre de Pyzo.

2. Le triangle de Sierpiński (TdS) est une figure géométrique auto-similaire composée de trois triangles de Sierpiński de taille deux fois plus petite.

On peut définir ses approximations comme suit. Le TdS de niveau 0 est un triangle équilatéral. Le TdS de niveau $n > 0$ est la réunion de trois copies de TdS de niveau $n - 1$, de taille deux fois plus petite et empilées comme sur la Fig. 1 (*relativement au sommet en bas à gauche*).

- (a) Écrire une fonction récursive `t2s(n, x, y, c)` qui dessine le triangle de Sierpiński de niveau n .

- (b) Dessiner le triangle de Sierpiński de niveau 2 (composé de 9 triangles).

Dessiner le triangle de Sierpiński de niveau 6 (Fig 2).

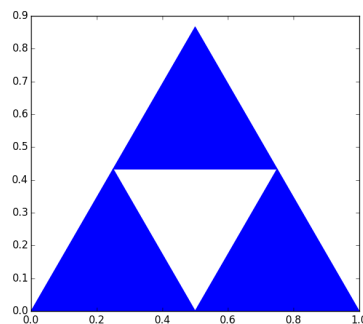


Fig. 1.

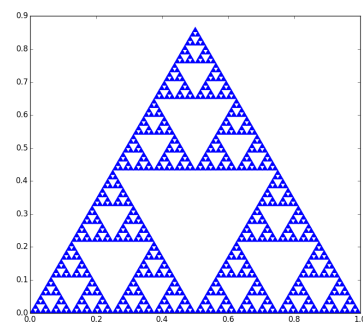


Fig. 2. Triangle de Sierpiński