

Récurtivité



Exercice 1

On considère la fonction factorielle $n!$ définie par $n! = 1 \times 2 \times \dots \times n$ si $n > 0$ et $0! = 1$.

1. Donner une définition récursive de la fonction mathématiques *factorielle*(n).
2. Écrire une fonction Python `fact(n)` qui implémente cette définition.
3. Dessiner l'arbre d'appels correspondant au calcul de `fact(4)`.

Exercice 2

La méthode du paysan russe est un très vieil algorithme de multiplication de deux nombres entiers déjà décrit sur un papyrus égyptien rédigé autour de 1650 avant J.-C. Il s'agissait de la principale méthode de multiplication en Europe avant l'introduction des chiffres arabes.

En notant $\bullet$ cette opération de multiplication russe, cet algorithme repose sur les relations :

$$x \bullet y = \begin{cases} 0 & \text{si } x = 0 \\ (x//2) \bullet (2y) & \text{si } x \text{ est pair} \\ (x//2) \bullet (2y) + y & \text{si } x \text{ est impair} \end{cases}$$

Écrire une fonction récursive `multiplication_russe(x,y)` qui calcule le produit de x et y en utilisant la méthode du paysan russe.

Exercice 3

On considère la suite u_n définie par la relation de récurrence suivante, où a et b sont des réels quelconques :

$$u_n = \begin{cases} a & \text{si } n = 0 \\ b & \text{si } n = 1 \\ 3u_{n-1} + 2u_{n-2} + 5 & \text{si } n \geq 2 \end{cases}$$

Écrire une fonction `suite(n,a,b)` qui renvoie le terme de rang n de cette suite pour des valeurs a et b données en paramètres.

Exercice 4

Écrire une fonction récursive `boucle(i,k)` qui affiche les entiers entre i et k .

Par exemple, `boucle(1,4)` doit afficher 1 2 3 4 (tout sur la même ligne).

Bien entendu, cette fonction ne doit pas utiliser de boucle `while` ou `for`.

Rappel : l'argument optionnel `end` dans la fonction `print` permet de changer le caractère final qui par défaut est `"\n"` (retour à la ligne).

Exercice 5

Écrire une fonction récursive `nombre_de_chiffres(n)` qui prend un entier positif ou nul n en argument et renvoie son nombre de chiffres.

Par exemple, `nombre_de_chiffres(12345)` doit renvoyer 5.

Exercice 6

En vous inspirant de l'exercice précédent, écrire une fonction récursive `nombre_de_bits_1(n)` qui prend un entier positif ou nul et renvoie le nombre de bits valant 1 dans la représentation binaire de n . Par exemple `nombre_de_bits_1(255)` doit renvoyer 8.

Remarque : cette fonction est utile dans de nombreux algorithmes bas niveau et porte souvent le nom de `popcount` (de l'anglais *population count*). Elle est souvent implémentée dans les unités arithmétiques et logiques des processeurs sur des entiers de taille fixe (32 ou 64 bits).

Exercice 7

Écrire une fonction récursive `appartient(v,t,i)` prenant en paramètres une valeur `v`, un tableau (une liste) `t` et un entier `i` et renvoyant `True` si `v` apparaît dans `t` entre l'indice `i` (inclus) et `len(t)` (exclu), et `False` sinon. Dans un premier temps on supposera que `i` est toujours compris entre 0 (inclus) et `len(t)` (exclu), puis on utilisera l'instruction `assert` pour restreindre les appels à la fonction `appartient(v,t,i)`.

Exercice 8 (Dichotomie)

On rappelle ici l'algorithme de recherche dichotomique dans sa version itérative vue en première, qui recherche l'indice de la valeur `val` dans le tableau d'entiers trié par ordre croissant `tab` :

```
def recherche_dichotomique(val,tab):
    a=0
    b=len(tab)
    while b - a > 1:
        c = (a + b) // 2 # c pour centre
        if val < tab[c]:
            b = c
        else:
            a = c
    if val == tab[a]:
        return a
    else:
        return -1
```

1. En s'inspirant du code itératif, définir une fonction récursive `dichotomie_rec(val,tab,a,b)` renvoyant l'indice `i` ($a \leq i < b$), s'il existe, tel que `tab[i]=val` ou `-1` sinon.

À titre d'aide, on pourra remarquer que :

- la présence de la boucle `while` dans le code itératif permet de déterminer un cas de base.
- les variables `a` et `b`, qui sont modifiées dans le code itératif, deviennent des paramètres dans la fonction récursive.

2. Écrire alors une fonction `dichotomie(val,tab)` permettant de faire une recherche dichotomique de `val` dans `tab`.

Exercice 9 (Figures fractales)

On souhaite utiliser le module `turtle` de Python pour réaliser quelques figures.

On rappelle ici quelques fonctions de base de ce module (Voir plus sur la [page de la librairie](#)) :

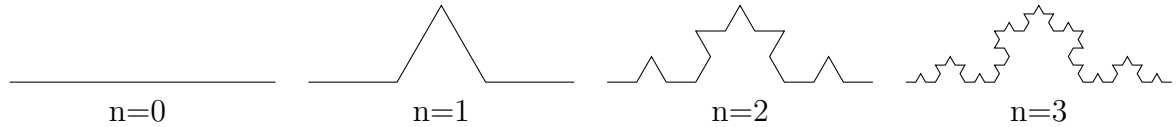
```
from turtle import * # importe toutes les fonctions du module
clearscreen() # effacer le contenu de l'écran
left(50) # tourner de 50 degrés sur la gauche
right(70) # tourner de 70 degrés sur la droite
forward(50) # avancer tout droit de 50 pixels
penup() # lever le crayon (pour ne pas écrire en se déplaçant)
pendown() # baisser le crayon (pour écrire en se déplaçant)
fillcolor('blue') # définir la couleur de remplissage
hideturtle() # cacher la tortue
showturtle() # montrer la tortue
begin_fill() # débiter une figure à remplir
end_fill() # remplir la figure
bye() # fermer la fenêtre (à faire dans l'interpréteur)
```

1. La courbe de Koch est une fractale imaginée par le mathématicien suédois Niels Fabian Helge von Koch en 1904.

On peut la créer à partir d'un segment de droite, en modifiant récursivement chaque segment de droite de la façon suivante :

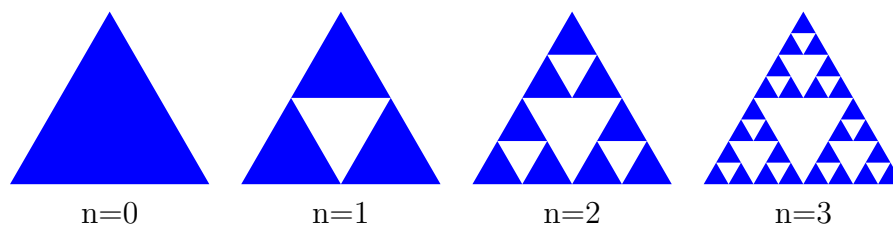
- On divise le segment de droite en trois segments de longueurs égales.
- On construit un triangle équilatéral sans base au-dessus du morceau central.

On répète ce processus n fois, n est appelé l'ordre.



Écrire une fonction `koch(n,1)` qui dessine avec le module `turtle` une courbe de Koch d'ordre n à partir d'un segment de longueur 1.

2. De manière similaire, définir une fonction `Sierpinski(n,1)` qui permet d'obtenir le triangle de Sierpinski (qu'il décrit en 1915) d'ordre n et de côté 1 comme ci-dessous :



Exercice 10 (Extrait modifié d'épreuve pratique)

Soit une image binaire représentée dans un tableau à 2 dimensions. Les éléments `M[i][j]` (i étant l'indice de ligne et j celui de colonne), appelés pixels, sont égaux soit à 0 soit à 1.

Une composante d'une image est un sous-ensemble de l'image constitué uniquement de 1 (ou uniquement de 0) qui sont côte à côte, soit horizontalement soit verticalement.

Par exemple, les composantes de

$$M = \begin{array}{|c|c|c|c|} \hline 0 & 0 & 1 & 0 \\ \hline 0 & 1 & 0 & 1 \\ \hline 1 & 1 & 1 & 0 \\ \hline 0 & 1 & 1 & 0 \\ \hline \end{array}$$

sont

$$M = \begin{array}{|c|c|c|c|} \hline 0 & 0 & 1 & 0 \\ \hline 0 & 1 & 0 & 1 \\ \hline 1 & 1 & 1 & 0 \\ \hline 0 & 1 & 1 & 0 \\ \hline \end{array}$$

On souhaite, à partir d'un pixel égal à 1 dans une image `M`, donner la valeur `val` à tous les pixels de la composante à laquelle appartient ce pixel.

La fonction `propager` prend pour paramètre une image `M`, deux entiers `i` et `j` et une valeur entière `val` (supposée différente de 1). Elle met la valeur `val` uniquement aux pixels de la composante du pixel `M[i][j]`, à condition que celui-ci vaille 1.

Par exemple, `propager(M,2,1,3)` donne

$$M = \begin{array}{|c|c|c|c|} \hline 0 & 0 & 1 & 0 \\ \hline 0 & 3 & 0 & 1 \\ \hline 3 & \mathbf{3} & 3 & 0 \\ \hline 0 & 3 & 3 & 0 \\ \hline \end{array}$$

Exemple :

```
>>> M = [[0,0,1,0],[0,1,0,1],[1,1,1,0],[0,1,1,0]]
>>> propager(M,2,1,3)
>>> M
[[0, 0, 1, 0], [0, 3, 0, 1], [3, 3, 3, 0], [0, 3, 3, 0]]
```

Compléter le code suivant de la fonction `propager(M,i,j,val)` :

```
def propager(M, i, j, val):
    if M[i][j] == ... :
        return
    else:
        M[i][j] = ...
        # l'élément en haut fait partie de la composante
        if ((i-1) >= 0 and M[i-1][j] == ...):
            propager(M, i-1, j, val)
        # l'élément en bas fait partie de la composante
        if ((...) < len(M) and M[i+1][j] == 1):
            propager(M, ..., j, val)
        # l'élément à gauche fait partie de la composante
        if ((...) >= 0 and M[i][j-1] == 1):
            propager(M, i, ..., val)
        # l'élément à droite fait partie de la composante
        if ((...) < len(M) and M[i][j+1] == 1):
            propager(M, i, ..., val)
```

Exercice 11 (Extrait BAC métropole candidats libres 2021 sujet 2)

On s'intéresse dans cet exercice à un algorithme de mélange des éléments d'une liste.

1. Pour la suite, il sera utile de disposer d'une fonction `echange` qui permet d'échanger dans une liste `lst` les éléments d'indice `i1` et `i2`.

Expliquer pourquoi le code Python ci-dessous ne réalise pas cet échange et en proposer une modification.

```
def échange(lst, i1, i2):
    lst[i2] = lst[i1]
    lst[i1] = lst[i2]
```

2. La documentation du module `random` de Python fournit les informations ci-dessous concernant la fonction `randint(a,b)` : Renvoie un entier aléatoire `N` tel que `a <= N <= b`.

Parmi les valeurs ci-dessous, lesquelles peuvent être renvoyées par l'appel `randint(0, 10)` ?

0 1 3.5 9 10 11

3. Le mélange de Fischer Yates est un algorithme permettant de permuter aléatoirement les éléments d'une liste. On donne ci-dessous une mise en œuvre récursive de cet algorithme en Python.

```
from random import randint

def melange(lst, ind):
    print(lst)
    if ind > 0:
        j = randint(0, ind)
        échange(lst, ind, j)
        melange(lst, ind-1)
```

- (a) Expliquer pourquoi la fonction `melange` se termine toujours.
- (b) Lors de l'appel de la fonction `melange`, la valeur du paramètre `ind` doit être égal au plus grand indice possible de la liste `lst`. Pour une liste de longueur `n`, quel est le nombre d'appels récursifs de la fonction `melange` effectués, sans compter l'appel initial ?
- (c) On considère le script ci-dessous :

```
lst = [v for v in range(5)]  
melange(lst, 4)
```

On suppose que les valeurs successivement renvoyées par la fonction `randint` sont 2, 1, 2 et 0. Les deux premiers affichages produits par l'instruction `print(lst)` de la fonction `melange` sont :

```
[0, 1, 2, 3, 4]  
[0, 1, 4, 3, 2]
```

Donner les affichages suivants produits par la fonction `melange`.

- (d) Proposer une version itérative du mélange de Fischer Yates.

Exercice 12 (Facultatif)

Écrire une fonction `permutation(liste)`, sous forme récursive, permettant d'obtenir l'ensemble des permutations possibles d'une liste `[e0, e1, ..., en]`.

Exemple :

```
assert permutation([1,2,3]) == [[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1]]
```