Il est possible d'utiliser `turtle` en ligne à : https://repl.it/languages/python_turtle

Exercice 1 (Division euclidienne)

Définition : Soient a et b deux entiers naturels avec $b \neq 0$. Il existe un unique couple d'entiers (q,r) tel que :

$$a = bq + r \quad avec \quad 0 \leqslant r < b$$

Les entiers q et r sont respectivement appelés le quotient et le reste de la division euclidienne de a par b.

1. Quels sont le reste et le quotient dans la division euclidienne de 157 par 7 ? Quelles sont les commandes `Python` permettant de les obtenir ?
2. On propose deux algorithmes récursifs pour le calcul du reste et du quotient de la division euclidienne sans utiliser les opérateurs `Python : % et //`. Vérifier que chacune des deux fonctions `Python` suivantes sont bien formées et calculent effectivement le reste et le quotient de la division euclidienne de `a` par `b` (entiers strictement positifs).

```
def reste_eucl(a : int, b : int) :
    """Calcul récursif du reste euclidien
    de a par b (entiers > 0)"""
    if a < b :
        return a
    return reste_eucl(a-b, b)
```

```
def quotient_eucl(a : int, b : int) :
    """Calcul récursif du quotient
    euclidien de a par b (entiers >0)"""
    if a < b :
        return 0
    return 1+quotient_eucl(a-b, b)
```

3. Ecrire sur le même modèle une fonction récursive `div_eucl(a:int,b:int,q:int)` renvoyant le reste `r` et le quotient `q` de la division euclidienne de `a` par `b`.

Exercice 2 (Recherche dichotomique)

1. Rappeler le principe de la recherche dichotomique.
2. Ecrire un algorithme récursif permettant de réaliser une recherche dichotomique dans une liste triée et l'implémenter en `Python`.

Exercice 3 (Les tours de Hanoï)

1. Chercher sur internet en quoi consiste le jeu des tours de Hanoï et décrire son principe.
2. Expliquer en quoi ce jeu peut être résolu par un algorithme récursif et donner cet algorithme.
3. Illustrer la résolution de ce jeu pour $n = 3$ (faire des dessins et indiquer les déplacements).

Exercice 4 (Un dessin récursif avec `turtle`)

On considère le code `Python` suivant :

```
from turtle import *

def CercleRec(x,y,r):
    up()
    goto(x,y)
    down()
    circle(r)
    if r>1:
        CercleRec(x+3*r/2,y+r/2,r/2)
        CercleRec(x,y+2*r,r/2)

speed("fastest")
CercleRec(0,0,64)
exitonclick()
```

1. Saisir ce script et le tester à l'aide de plusieurs appels de la fonction `CercleRec()` et décrire ce qu'il fait (faire un dessin et numéroter les cercles dans ordre dans lequel ils sont tracés).
2. Quel argument de la fonction `CerclesRec` permet d'assurer qu'elle n'est appelée qu'un nombre fini de fois ?
3. **Mini-projet** : modifier ce script afin de compléter le tracé pour que le cercle initial soit entouré de quatre cercles plus petits. Ajouter des couleurs.

Exercice 5 (Calcul de puissance)

1. On rappelle que pour calculer x^n où x est un nombre quelconque et n un entier positif, on multiplie x par lui-même $n-1$ fois. Par exemple $x^2 = x \times x$,$x^3 = x \times x \times x, \cdots$ De plus, pour $n = 0$ on a toujours $x^0 = 1$.
   (a) Proposer une fonction récursive en `Python` nommée `puissance(x,n)` permettant de calculer x^n de façon récursive (donc sans utiliser l'opérateur dédié $**$).
   (b) Préciser les spécifications de cette fonction. (ajouter une chaîne de documentation à la fonction) puis ajouter des assertions pour contrôler la bonne utilisation de cette fonction.
   (c) Vérifier que cette fonction récursive est bien formée.
   (d) Dresser l'arbre d'appels produit à l'exécution de `puissance(2,5)`. En déduire la taille maximale de la pile d'exécution.
2. L'algorithme *d'exponentiation rapide* permet de calculer la n-ième puissance d'un nombre de façon efficace en reposant sur les égalités suivantes :
   $$\text{Pour tout } x \in \mathbb{R}, \text{ pour tout } n \in \mathbb{N}, \quad x^{2n} = (x^2)^n \text{ et } x^{2n+1} = x(x^2)^n$$
   (a) Combien de multiplications sont nécessaires au calcul de 2^{65} avec cette méthode.
   (b) Écrire un algorithme récursif d'exponentiation rapide et l'implémenter en `Python`.

Exercice 6 (Suite de Fibonacci)

1. Ecrire un algorithme récursif `fibo(n)` permettant de calculer le $n^{\text{ème}}$ terme de la suite de Fibonacci c'est à dire la suite (u_n) définie par $u_0 = u_1 = 1$ et $u_{n+2} = u_{n+1} + u_n$ pour tout $n \geqslant 0$.
2. Ecrire la suite des appels récursifs nécessaires au calcul de u_5 avec leurs imbrications (faire un arbre d'appels).
   En déduire une raison pour laquelle cet algorithme récursif n'est pas efficace.
3. On propose le code suivant écrit en `Python` :
   Écrire la suite des appels récursifs produite à l'exécution de l'instruction `fibo_acc(5,1,1)` avec leurs imbrications. Que constate-t-on ?

```
def fibo_acc(n, a, b) :
    """Calcul récursif du n-ième terme
    de la suite de Fibonacci
    avec accumulateurs :
    a initialisé à 1 et b à 1"""
    if n == 0 :
        return a
    elif n == 1 :
        return b
    else :
        return fibo_acc(n-1, b, a+b)
```

Exercice 7 (Mini-projet : d'autres fractales avec `turtle`) Chercher sur internet ce que sont que les triangles de Sierpinski, les flocons de Von Koch et les courbes du dragon. Choisir un de ces motifs et le programmer récursivement avec `turtle`.

Exercice 8 (Mini-projet : quickSort) Ecrire un algorithme récursif permettant de trier un tableau par dichotomie : il s'agit de choisir une valeur pivot (au départ le dernier élément du tableau par exemple) et de construire le tableau trié en concaténant le sous-tableau trié constitué des valeurs inférieures au pivot, le pivot et le sous-tableau trié constitué des valeurs supérieures au pivot (le pivot est donc inséré à sa place). Le programmer en `Python`.