TD n° 2 : Récursivité

Exercice 1. *Une suite*
On considère la suite (u_n) définie par $u_0 = 5$ et, pour tout $n \in \mathbb{N}$, $u_{n+1} = 1 - 2u_n$.
Écrire une fonction récursive d'entête

```
def u(n: int) -> int:
```

qui renvoie la valeur de u_n.

Exercice 2. *Calcul de somme*
Écrire une fonction récursive d'entête

```
def somme(tab: [float]) -> float:
```

qui renvoie la valeur de la somme des éléments contenus dans `tab`.

Exercice 3. *Moyenne arithmético-géométrique*
Soient a et b deux réels positifs. On définit les deux suites positives (u_n) et (v_n) par $u_0 = a$, $v_0 = b$ et la relation de récurrence

$$\forall n \in \mathbb{N},\ u_{n+1} = \sqrt{u_n v_n},\ v_{n+1} = \frac{u_n + v_n}{2}.$$

1. Écrire une fonction itérative d'entête

   ```
   def ari_geo(a: float, b: float, n: int) -> float:
   ```

   qui renvoie le couple (a_n, b_n).
2. Même question mais avec une fonction récursive.

Exercice 4. *Calcul de puissances*
On souhaite calculer la puissance (entière positive) d'un nombre en n'utilisant que des multiplications (et donc pas l'opérateur `**`).

1. On observe que, pour tout $x \in \mathbb{R}$ et $n \in \mathbb{N}$, $x^{n+1} = x \times x^n$.
   a) Écrire une fonction récursive d'entête

   ```
   def puissance(x: float, n: int) -> float:
   ```

   qui renvoie la valeur de x^n basée sur l'observation ci-dessus.
   b) Démontrer la terminaison de cette fonction.
   c) Estimer le nombre $C(n)$ de multiplications effectuées lors de l'appel `puissance(x, n)`.
2. On observe maintenant que, pour tout $x \in \mathbb{R}$ et $n \in \mathbb{N}$, $x^n = \begin{cases} x^{\frac{n}{2}} \times x^{\frac{n}{2}} & \text{si } n \text{ est pair,} \\ x \times x^{\frac{n-1}{2}} \times x^{\frac{n-1}{2}} & \text{si } n \text{ est impair.} \end{cases}$
   a) Écrire une fonction récursive d'entête

   ```
   puissance_rapide(x: float, n: int) -> float:
   ```

   qui renvoie la valeur de x^n basée sur cette nouvelle observation.
   b) Estimer le nombre $D(n)$ de multiplications effectuées lors de l'appel `puissance_rapide(x, n)`.

Exercice 5. *Coefficients binomiaux*
En utilisant la formule du triangle de Pascal :

$$\forall n \in \mathbb{N},\ \forall k \in [\![1\,;n-1]\!],\ \binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1},$$

écrire une fonction récursive d'entête

```
def binom(n: int, k: int) -> int:
```

qui renvoie la valeur de $\binom{n}{k}$.

Exercice 6. *Méthode de Hörner, d'après CCP 2020*
On considère un polynôme $P = \sum_{k=0}^{n} a_k X^k \in \mathbb{R}_n[X]$. Celui-ci est représenté en Python par la liste des coefficients `coeff = [` a_0`, ...,` a_n`]`.

1. Écrire une fonction d'entête

   ```
   def eval(coeff: [float], b: float) -> float:
   ```

   où la liste `coeff` est celle des coefficients de P, qui renvoie la valeur de $P(b)$ en la calculant via la formule naturelle $P(b) = \sum_{k=0}^{n} a_k b^k$.

En admettant que le calcul de b^k utilise $k-1$ multiplications (cf Q1 de l'exercice 4), on trouve une complexité quadratique en le degré n du polynôme.
On peut être plus astucieux en utilisant l'algorithme de Hörner qui se base sur l'égalité suivante :

$$P(X) = ((\cdots((a_n X + a_{n-1})\,X + a_{n-2})\,X + \ldots)\,X + a_1)\,X + a_0.$$

Autrement dit, pour évaluer P en b, on commence par calculer $a_n \times b + a_{n-1}$ puis on multiplie le résultat par b et on ajoute a_{n-2}, ensuite on multiplie ce résultat par b et on ajoute a_{n-3}, etc.

2. a) Écrire une fonction itérative d'entête

   ```
   def horner1(coeff: [float], b: float) -> float:
   ```

   où `coeff` est la liste des coefficients de P, qui renvoie la valeur de $P(b)$ en utilisant l'algorithme de Hörner.

   b) Quelle est la complexité de `horner1` en fonction du degré n du polynôme ?

3. a) Écrire une fonction récursive d'en-tête

   ```
   def horner2(coeff: [float], b: float) -> float:
   ```

   où `coeff` est la liste des coefficients de P, qui renvoie la valeur de $P(b)$ en utilisant l'algorithme de Hörner.

   b) Quelle est la complexité de `horner2` en fonction du degré n du polynôme ?

Exercice 7. *Occurrences*
Écrire une fonction récursive d'entête

```
def occurrences(mot: str, lettre: str) -> int:
```

qui renvoie le nombre d'occurrences de `lettre` (chaîne composée d'un unique caractère) dans `mot`.
Par exemple

```
>>> occurrences("informatique", 'i')
2
>>> occurrences("informatique", 'z')
0
```

Exercice 8. *Palindromes*
Un palindrome est un mot qui se lit indifféremment de gauche à droite ou de droite à gauche comme ICI, KAYAK ou RESSASSER.
Écrire une fonction récursive d'entête :

```
def est_palindrome(mot: str) -> bool:
```

qui renvoie `True` si `mot` est un palindrome et `False` sinon.
On pourra aussi faire l'épreuve 3.01 du challenge.

Exercice 9. *Dichotomie*
On suppose que l'on dispose d'une liste de flottants `tab` triée par ordre croissant. On cherche si un flottant `x` appartient ou non à cette liste. Afin que cette recherche soit rapide (complexité logarithmique en la taille de la liste), on procède par dichotomie.

1. Écrire une fonction récursive d'entête :

   ```
   def dicho(x: float, tab: [float], deb: int, fin: int) -> bool:
   ```

   qui renvoie `True` ou `False` suivant que `x` se trouve ou non dans la liste `tab` entre les indices `deb` (inclus) et `fin` (exclu).

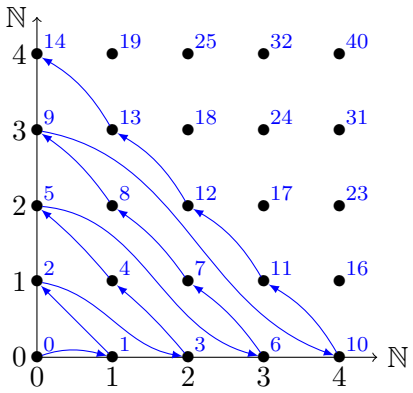
⋆ 2. Modifier cette fonction afin qu'elle renvoie désormais le nombre d'appels récursifs effectués.

⋆ **Exercice 10.** *Numéroter les éléments de* $\mathbb{N}^2$
Il est possible de numéroter chaque élément $(i, j) \in \mathbb{N} \times \mathbb{N}$ selon le procédé suggéré ci-contre : $(0, 0)$ est numéroté par 0 puis $(1, 0)$ par 1, $(0, 1)$ par 2, $(0, 2)$ par 3, $(1, 1)$ par 4, etc.
Écrire une fonction récursive d'entête

```
def numero(i: int, j: int) -> int:
```

qui renvoie le numéro du point de coordonnées (i, j) selon ce principe de numérotation.

Remarque mathématique : cela montre que $\mathbb{N}^2$ *est dénombrable. La démonstration historique de Cantor donne même une formule explicite*[a] *pour le numéro associé à* (i, j) *en fonction de* i *et* j*.*



a. cf `https://fr.wikipedia.org/wiki/Fonction_de_couplage`

⋆ **Exercice 11.** *Fonction 91 de McCarthy*
On considère la fonction suivante :

```
1 def mccarthy(n: int) -> int:
2     if n > 100:
3         return n - 10
4     else:
5         return mccarthy(mccarthy(n + 11))
```

1. Que renvoie `mccarthy(k)` si k est un entier strictement plus grand que 100 ?
2. Et si $90 \leqslant k \leqslant 100$?
3. Et pour $0 \leqslant k \leqslant 89$?

⋆ **Exercice 12.** Faire l'épreuve 5.05 du challenge.

TP **Exercice 13.** *Fractales : flocon de Koch et triangle de Sierpinski*

On utilise ici le module `turtle` de Python qui permet de dessiner. On peut trouver une documentation complèteen ligne mais pour cet exercice, les deux commandes suivantes suffisent :

`forward(longueur)` : avance le stylo d'une `longueur` donnée (selon l'angle où se trouve le stylo) ;

`left(angle)` ou `right(angle)` : tourne le stylo vers la gauche ou la droite d'un `angle` donné.

Pour dessiner la *courbe de Koch*, on part d'un segment, on le découpe en trois morceaux égaux, on construit au-dessus du morceau central un triangle équilatéral sans base et on réitère le processus sur chacun des segments de la nouvelle figure et ainsi de suite. Le nombre de répétition est appelé l'ordre.

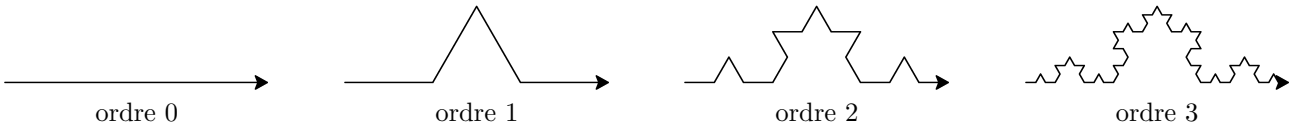


FIGURE 1 – Courbe de Koch pour différents ordres

En pratique, à l'ordre 0 (cas d'arrêt) un simple `forward` suffit ; à l'ordre 1, il y a quatre `forward` entrecoupés de `left` et `right` pour changer de directions, etc. Ce processus se prête donc bien à la récursivité.

1. Écrire une fonction récursive `courbe_Koch` qui :
   - prend en argument un entier `longueur` représentant la longueur du segment de départ et un entier `n` représentant l'ordre ;
   - permet de dessiner la courbe de Koch d'ordre n.

Ce principe est souvent appliqué avec au départ un triangle équilatéral au lieu d'un simple segment. Le *flocon de Koch* est la limite des courbes obtenues lorsqu'on répète indéfiniment le processus. Cette courbe a la particularité d'être de longueur infinie tout en délimitant une portion du plan d'aire finie.

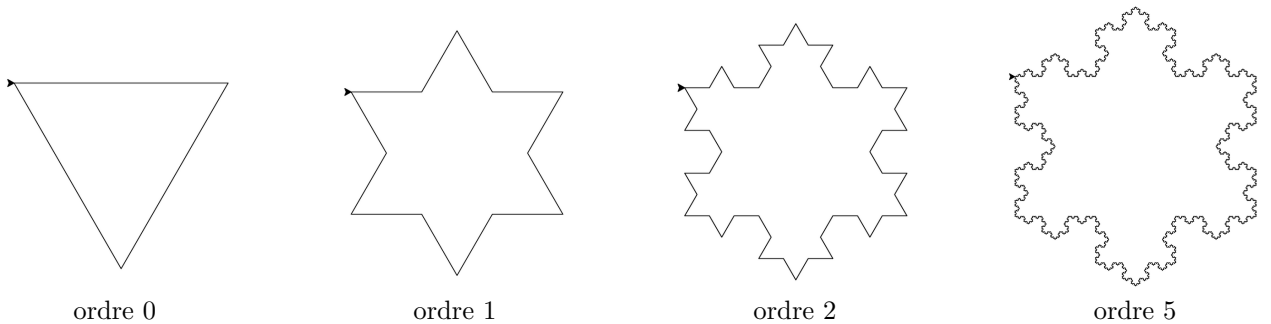


FIGURE 2 – Flocon de Koch pour différents ordres

2. Écrire une fonction récursive `flocon_Koch` qui :
   - prend en argument un entier `taille` représentant la longueur du côté du triangle de départ et un entier `n` représentant l'ordre ;
   - permet de dessiner le flocon de Koch d'ordre n ;
   - fait appel à la fonction `courbe_Koch` pour chacun des trois côtés du triangle de départ.

⋆ 3. Écrire une fonction qui permet de tracer le triangle de Sierpinski à l'aide de `turtle`.

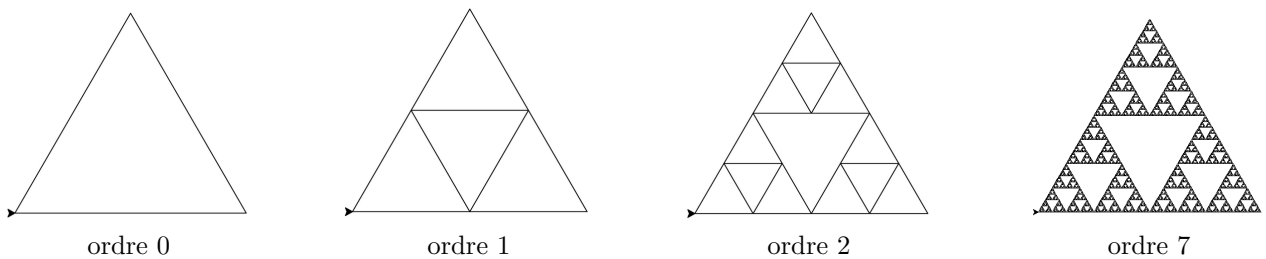


FIGURE 3 – Triangle de Sierpinski pour différents ordres