L'usage de la calculatrice est interdit.

Les résultats des questions pourront être réutilisés ainsi que les différentes fonctions demandées non traitées au cours du problème.

Le sujet comporte deux problèmes indépendants ainsi qu'une annexe présentant entre autres un certain nombre de fonctions utiles à la rédaction des programmes en langage Python.

Problème 1

Les systèmes de Lindenmayer, appelés aussi **L-Systèmes**, ont été imaginés par le biologiste A. Lindenmayer et modélisent le processus de développement et de prolifération de plantes. Le concept central des L-Systèmes est la représentation d'une plante par une chaîne de caractères. Cela permet de modéliser son évolution, voire sa destruction par des agents pathogènes, au moyen de règles de transformations de ces caractères.

Dans tout le problème on se place dans un plan orienté muni d'un repère (non visible). L'axe des abscisses est dirigé classiquement vers l'est (**azimut** 0 degrés) et l'axe des ordonnées vers le nord (azimut 90 degrés).

Partie A : représentation des L-Systèmes

Le module turtle

Le langage `Python` dispose d'un module `turtle` permettant de réaliser des figures en déplaçant une « tortue » symbolisée par un triangle. Au départ, la tortue est tournée vers l'est et l'unité de longueur est le pixel. Trois instructions seront utiles (x et a pouvant être de type entier ou flottant) :

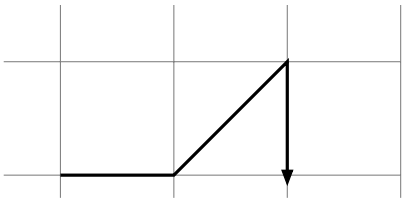
- `forward`(x) : fait avancer la tortue de x pixels
- `left`(a) et `right`(a) : font tourner la tortue respectivement de a degrés vers sa gauche ou vers sa droite (la tortue n'avance pas).

Par exemple le code suivant :

```
from turtle import *
from math import sqrt

forward(100)
left(45)
forward(100*sqrt(2))
right(135)
forward(100)
```

trace à l'écran :



1. Écrire un programme `Python` permettant de réaliser la figure « maison » ci-après avec le mode `turtle`, sachant que :

$$AB = BC = CD = DE = EA = 100,\ (AB) \perp (AE) \text{ et } (AB) \perp (BC).$$

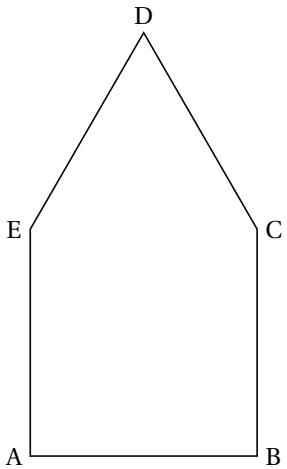


Figure « maison »

Un alphabet pour coder les figures

Comme on réalise régulièrement les mêmes instructions, on se propose de **décrire** les figures selon les règles suivantes : **une figure** $\mathcal{F}$ est définie par la donnée d'un triplet contenant :

- une longueur ℓ
- un pas de rotation a, donné en degré
- un mot, appelé **motif**, sur l'alphabet $\{\texttt{F}, \texttt{+}, \texttt{-}\}$ avec comme convention :
  - `F` : avancer de ℓ
  - `+` : tourner à gauche de l'angle défini par le pas de rotation
  - `-` : tourner à droite de l'angle défini par le pas de rotation.

Par exemple, la figure $\mathcal{F}(50, 90, \texttt{F+F+F+F})$ représente un carré de 50 unités de côté.

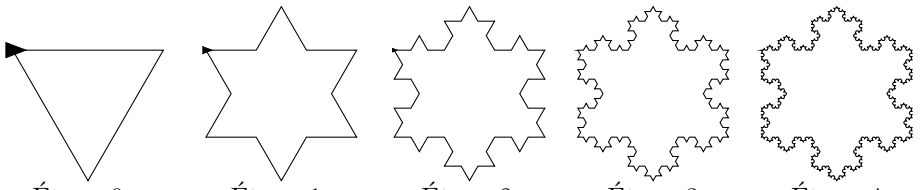
2. Décrire la figure « maison » sous la forme $\mathcal{F}(\ell, a, m)$ en précisant les valeurs de ℓ, a et m (on commencera du point A).

3. Écrire une fonction `dessiner` qui :

- reçoit en entrée :
  - `unite` : un nombre représentant la longueur ℓ
  - `angle` : un nombre représentant le pas de rotation a
  - `motif` : le motif m de la figure sous forme d'une chaîne de caractères
- affiche le dessin $\mathcal{F}(\ell, a, m)$ et retourne 0 si celui-ci a pu être réalisé intégralement et 1 sinon (par exemple si un caractère non défini est présent dans le motif).

Les L-Systèmes

L'intérêt des L-Systèmes est de permettre de décrire simplement l'évolution d'une figure. Prenons l'exemple du flocon de Von Koch (l'échelle d'une image à l'autre a été ajustée pour une meilleure visibilité) :


Étape 0 Étape 1 Étape 2 Étape 3 Étape 4

En choisissant un pas de rotation de 60°, les motifs des deux premières figures sont :

Étape 0 : `F -- F -- F`

Étape 1 : `F+F--F+F` `--` `F+F--F+F` `--` `F+F--F+F`

À chaque étape, chaque lettre `F` dénotant un segment est remplacé par le motif `F+F--F+F`. Un L-système est la donnée d'un **axiome** (motif de départ) et d'une **règle** ou d'un ensemble de règles. Dans notre exemple, l'axiome est `F--F--F` et la règle `F`→`F+F--F+F`.

Dans la suite, on considère un L-système avec un axiome `F` et une seule règle `F`→ m où m est une chaîne de caractères, appelée **membre droit** de la règle.

4. Écrire une fonction `suivant` qui :
- reçoit en entrée deux chaînes de caractères :
  - `motif` : le motif de la figure à une étape donnée
  - `regle` : le membre droit de la règle
- retourne en sortie une chaîne de caractères représentant la figure à l'étape suivante.

Par exemple :

```
>>> suivant('F--F--F','F+F--F+F')
'F+F--F+F--F+F--F+F--F+F--F+F'
```

5. Programmer la fonction `evolution` qui :
- reçoit en entrée :
  - `axiome` : une chaîne de caractères représentant le motif de départ
  - `regle` : une chaîne de caractères indiquant le membre droit de la règle de mutation
  - `etape` : un entier indiquant le numéro de l'étape à calculer
- retourne en sortie une chaîne de caractères représentant la figure à l'étape demandée.

Par exemple :

```
>>> evolution('F','F+',4)
'F++++'
```

6. Démontrer qu'à l'étape n ($n \in \mathbb{N}$), la chaîne de caractères représentant le flocon contient 3×4^n caractères `F` et 4^{n+1} caractères de rotation (`+` ou `-`).

7. La tortue trace à la vitesse de 1000 pas par seconde et tourne à la vitesse de 800 degrés par seconde. Quel temps mettra-t-elle pour dessiner le tracé du flocon de l'étape 4 si l'on donne pour longueur d'un segment $\ell = 2$?

Nouveau mode de représentation

La durée de tracé étant assez longue avec le module `turtle`, on se propose d'utiliser à partir de maintenant le module `pyplot` du package `matplotlib` pour effectuer le tracé.

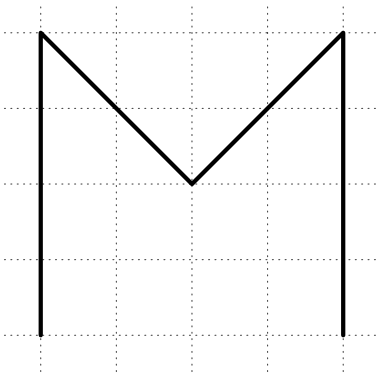
La fonction `plot` reçoit trois arguments : deux listes X et Y de même taille, dont on notera x_i et y_i les éléments, et un troisième paramètre représentant le style de tracé sous forme d'une chaîne de caractère. Pour une ligne noire continue, ce style est décrit par la chaîne `k-`. Elle a pour effet de tracer la ligne brisée reliant les points de coordonnées (x_i, y_i).

Le code

```
from matplotlib.pyplot import *

X = [0, 0, 2, 4, 4]
Y = [0, 4, 2, 4, 0]
plot(X, Y, "k-")
show()
```

affiche à l'écran :



8. Écrire un programme permettant de tracer la figure « maison » à l'aide du module `pyplot` pour une longueur $AB = 4$.

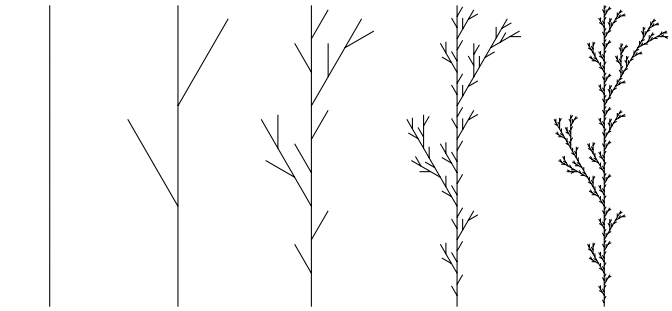
En se plaçant dans un repère orthonormé $(O, \vec{i}, \vec{j})$, on note (x, y) les coordonnées de la tortue et d son azimut en degrés. Ces 3 quantités pourront être implémentées par des flottants.

9. En considérant une longueur ℓ, un pas de rotation de a degrés et un motif m, exprimer les nouvelles valeurs de x, y et d en fonction des anciennes lorsque l'on rencontre le caractère `F` dans le motif. Faire de même lorsque l'on rencontre le caractère `+`.

10. Écrire une fonction `dessine` qui a le même effet que la fonction `dessiner` de la question 3 mais qui utilise le module `matplotlib` (et non le module `turtle`).

Gestion des ramifications

On souhaite à présent représenter une plante à l'aide d'un L-Système. Par exemple, avec une règle préalablement choisie et partant d'une branche `F`, on peut obtenir les images suivantes où le pas de rotation est de 20° (l'échelle est ajustée d'une image à l'autre et l'azimut de départ initialisé à 90° pour un rendu plus réaliste) :



Étape 0 Étape 1 Étape 2 Étape 3 Étape 4

11. Écrire les 20 premiers caractères d'une règle de transformation de l'exemple ci-dessus.

On se propose d'ajouter deux nouveaux symboles à l'alphabet des L-Systèmes :
- `[` : place l'état de la « tortue » (coordonnées et orientation) en tête d'une pile
- `]` : dépile la dernière position de la « tortue » et replace la tortue à cet endroit (sans effectuer de tracé). La pile sera gérée par une liste et par les méthodes `append` et `pop`.

L'exemple pourrait alors être construit par la règle `F` → `F[+F]F[-F]F`.

12. Quel avantage présente l'ajout de ces deux nouveaux symboles ?

13. Dessiner la figure $\mathcal{F}(2, 90, \texttt{F[-F[+F]-F]F})$ où la longueur est donnée en centimètres et le pas de rotation en degrés. L'azimut initial est 90.

14. Réécrire la fonction `dessineAvecPile` pour qu'elle tienne compte de ces deux nouveaux symboles ; son interface devra accepter comme paramètres :
- `unite` : la longueur ℓ
- `angle` : le pas de rotation a en degrés
- `motif` : une chaîne de caractères m représentant le motif de la figure
- `azimut` : l'azimut initial du tracé (0 par défaut afin d'assurer une cohérence avec les fonctions du mode `turtle` et `matplotlib`).

Partie B : génération automatique de L-Systèmes

Dans cette seconde partie du problème, on cherche à générer des règles de transformation pour obtenir des L-Systèmes ayant des allures de plantes. L'axiome sera systématiquement `F` et la direction de départ 90°.

Le choix retenu ici est le recours à un algorithme (dit **génétique**) dont le principe est le suivant : partant d'une population souche de 100 individus représentant 100 règles générées de manière aléatoire, on répète les opérations suivantes (qui seront détaillées par la suite) un grand nombre de fois :
- **Sélection** : on conserve les 80 plus belles plantes obtenues,

– **Croisement** : parmi les individus restants, on en choisit 40 pour se « reproduire » deux par deux et ainsi générer 20 descendants,
– **Mutation** : certains individus subissent une légère mutation.

On ne se propose pas dans ce problème de réaliser l'intégralité de l'algorithme génétique mais on s'intéresse à certaines étapes.

Génération de la population d'origine

On souhaite réaliser une fonction `genereRegle` qui ne reçoit aucun argument. Cette fonction renverra une chaîne composée de 15 à 30 caractères aléatoires, représentant une règle de transformation d'un L-Système.

Voici une proposition naïve de fonction :

```
from random import *

def genereRegle():
    """
    Fonction qui ne reçoit pas d'argument et retourne une règle sous
    forme d'une chaîne de caractères
    """
    alphabet = ['F', '-', '+', '[', ']']
    regle = ""
    for i in range(randint(15, 30)):
        regle = regle + choice(alphabet)
    return regle
```

1. La chaîne générée ici est totalement aléatoire et peut ne pas être le motif d'une figure. Par exemple la chaîne `]+F-F][---` représente une règle **invalide**. En effet, au premier symbole `]`, la pile dont le principe a été présenté en fin de partie A est vide. Écrire une fonction `verifie` qui indique si la chaîne reçue en argument est une règle de transformation valide.

2. Pour une chaîne représentant une règle valide, un certain nombre de symboles peuvent être inutiles. C'est ainsi que la chaîne : `F+-[F-F]+F[F-]-F` peut se simplifier en `F[F-F]+F[F]-F` (suppression de trois caractères inutiles car sans effet sur le dessin). Voici une proposition de fonction (comportant des erreurs) qui reçoit une règle sous forme d'une chaîne de caractères et retourne une chaîne simplifiée en se limitant à la suppression des motifs `+-`, `-+`, `+]` et `-]` :

```
1  def simplifie(regle) :
2      """
3      Fonction qui reçoit une règle sous forme d'une chaîne de caractères et
4      retourne une chaîne de caractères représentant la règle simplifiée.
5      """
6      i, reponse = 0, ""
7      while i < len(regle):
8          double = regle[i] + regle[i+1]
9          if double == "+-" or double == "-+":
10             i = i + 1
11         elif double != "-]" or double != "+]":
12             reponse = reponse + regle[i]
13         i = i + 1
14     reponse = reponse + regle[-1]
15     if len(reponse) != len(regle):
16         reponse = simplifie(reponse)
17     return reponse
```

a. À quoi servent les lignes 15 et 16 de cette fonction ?

b. Corriger les erreurs qui empêchent cette fonction de réaliser le travail demandé.

3. Écrire une fonction `population` qui reçoit un entier `n` et renvoie une liste de `n` règles simplifiées, deux à deux distinctes et valides comportant au moins 3 symboles `[` et 2 symboles de rotation.

Mutation

La mutation d'un individu va consister ici en l'échange d'un symbole `+` de la règle en un symbole `-` ou inversement. Par exemple la règle `F+F-F` pourrait muter en `F-F-F`.

4. Réaliser une fonction `mutation` qui reçoit une chaîne de caractères représentant une règle et renvoie une nouvelle règle ayant subi une mutation quelconque : un des symboles de rotation de la chaîne, choisi aléatoirement, sera transformé en son symbole opposé.

5. La fonction `mutationPopulation` reçoit une liste de règles et un nombre `p` $\in [0, 1]$ puis renvoie une nouvelle liste de règles :

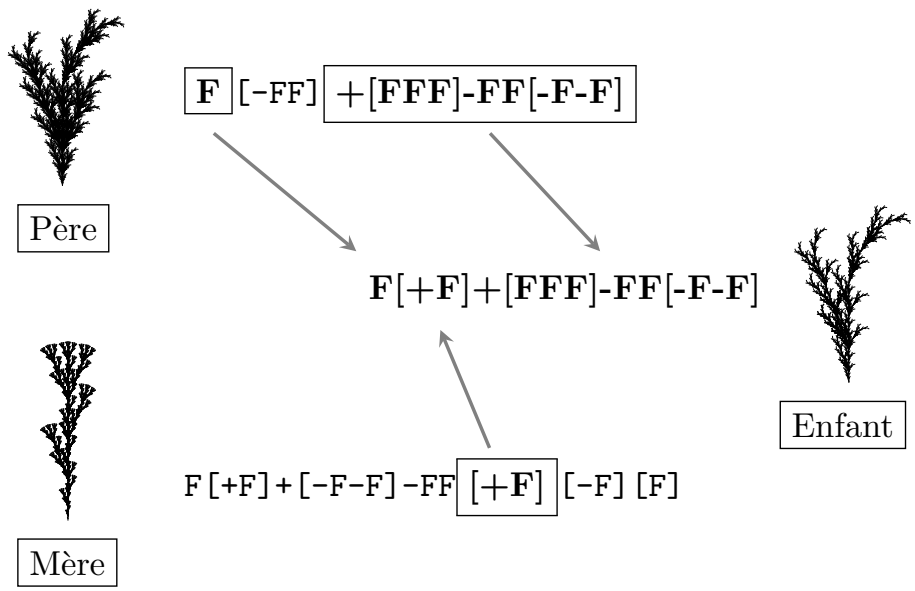
```
def mutationPopulation(L, p) :
    """
    L est une liste de règles et p un nombre entre 0 et 1
    """
    for i in range(len(L)) :
        if random() < p :
            L[i] = mutation(L[i])
    return L  # La liste en argument est modifiée et retournée.
```

a. Justifier qu'à l'appel de la fonction, chaque règle de la liste subit une mutation avec une probabilité de p.

b. Pour une population de 100 règles, combien de règles en moyenne sont modifiées (justifier la réponse).

Croisement

On souhaite à présent réaliser le croisement de deux L-Systèmes. On appelle **branche** d'une règle tout motif situé entre deux crochets. Le principe retenu est le suivant : fabriquer une nouvelle règle en remplaçant une branche de la première règle par une branche de la seconde comme l'illustre l'exemple ci-dessous :



6. Écrire une fonction `extraitBranche` qui reçoit une règle valide donnée sous forme d'une chaîne de caractères et renvoie un triplet contenant cette chaîne coupée en trois, l'élément central étant une branche choisie de manière aléatoire. On pourra par exemple choisir aléatoirement un symbole `[` et extraire la branche associée en trouvant le caractère `]` correspondant.
Par exemple :

```
>>> extraitBranche('F[-FF]+[FF[-F]]-FF[-F-F]')
('F', '[-FF]', '+[FF[-F]]-FF[-F-F]')
>>> extraitBranche('F[-FF]+[FF[-F]]-FF[-F-F]')
('F[-FF]+[FF', '[-F]', ']-FF[-F-F]')
```

7. Écrire alors une fonction `croise(r1,r2)` qui, recevant deux règles, renvoie (sous forme d'une chaîne de caractères), un enfant issu de ces deux règles.

Problème 2

Notations

- Dans ce problème, n et k désigneront des entiers naturels non nuls.
- $[\![0, n]\!]$ désignera l'ensemble des entiers compris au sens large entre 0 et n.
- La notation $\log_2$ désignera le logarithme de base 2, c'est-à-dire $\forall x > 0,\ \log_2(x) = \dfrac{\ln(x)}{\ln(2)}$ où ln est la fonction logarithme népérien.

Préambule

Par définition, tout ensemble fini est appelé **alphabet** et ses éléments sont appelés **lettres**.
Un **mot** sur l'alphabet $\mathcal{A}$ est une concaténation de lettres de $\mathcal{A}$.
La concaténation des n lettres a_1, a_2,..., a_n est notée $a_1a_2a_3\ldots a_{n-1}a_n$. La **longueur d'un mot** est égale au nombre de lettres composant ce mot. L'ensemble des mots de longueur n sur l'alphabet $\mathcal{A}$ est noté $\mathcal{A}^n$. L'ensemble des mots sur l'alphabet $\mathcal{A}$ est noté $\mathcal{A}^*$. Un mot sur l'alphabet $\{0, 1\}$ est appelé **mot binaire**.

Partie A

1. Énumérer les mots binaires de longueur 3.

2. La fonction `product` du module `itertools` permet de générer les éléments d'un produit cartésien.
Ainsi, la commande `product(A, repeat=n)` permet d'itérer sur la liste des éléments de $\mathcal{A}^n$. Dans l'exemple suivant, les mots binaires de longueur 2 sont affichés. La liste `[0,1]` correspond à l'alphabet et la longueur des mots souhaités est précisée dans `repeat`.

```
>>> from itertools import product
>>> liste = []
>>> for u in product([0,1], repeat = 2):
...         liste.append(u)
...
>>> liste
[(0, 0),(0, 1),(1, 0),(1, 1)]
```

Nous noterons $\boldsymbol{M(n,k)}$ l'ensemble des mots de longueur n sur l'alphabet $[\![0, k-1]\!]$.

Écrire une fonction `motn` donnant les mots de $M(n,k)$ pour $k \leq 10$. La fonction prendra comme paramètres n et k et renverra une liste contenant les mots sous forme de chaînes de caractères.

3. Combien y a-t-il d'éléments dans $M(n,k)$?

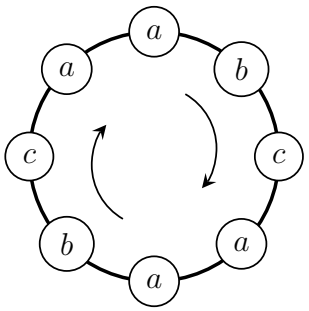
4. En supposant qu'une liste puisse contenir jusqu'à 500 000 000 éléments, quelle longueur maximale peut-on donner aux mots binaires pour que la fonction `motn` s'exécute sans erreur (c'est-à-dire $k = 2$) ? Vous pourrez vous aider de la courbe donnée en annexe.

Partie B

Un **mot circulaire** est une séquence $(a_1 a_2 \dots a_n)$ de lettres données avec un ordre circulaire, c'est-à-dire que la lettre a_1 suit a_n.

Les mots $\begin{cases} a_1 a_2 \dots a_n, \\ a_2 \dots a_n a_1, \\ a_3 \dots a_1 a_2, \\ \vdots \\ a_n a_1 \dots a_{n-1} \end{cases}$ représentent le même mot circulaire.

Mot circulaire de longueur 8



$(abcaabca) = (bcaabcaa) = \dots$

La **longueur d'un mot circulaire** est égale à la longueur de n'importe lequel de ses représentants. Par exemple, les mots 101, 011 et 110 sont les représentants du même mot circulaire de longueur 3 c'est-à-dire :

$$(101) = (011) = (110).$$

1. Déterminer tous les représentants possibles du mot circulaire (1011).

2. Déterminer le nombre maximal de représentants d'un mot circulaire de n lettres. La réponse devra être justifiée.

3. Déduire de la question précédente un minorant du nombre de mots circulaires différents obtenus à partir de $M(n,k)$. La réponse devra être justifiée.

4. On se propose de définir une classe (au sens du langage Python) `MotCirculaire` avec :
- un constructeur initialisant une instance à partir d'une chaîne de caractères donnant un représentant du mot circulaire,
- deux attributs `chaine` et `longueur` stockant d'une part le représentant concaténé à lui-même et d'autre part la longueur du représentant,
- une méthode `representant` sans paramètre et retournant le représentant initial,
- une méthode `estEgal` permettant de déterminer si deux instances de cette classe représentent ou non le même mot circulaire. Cette méthode aura comme paramètre un mot circulaire et renverra un booléen, de sorte qu'on pourra écrire : `if mot1.estEgal(mot2):...`

Le squelette de la classe `MotCirculaire` est proposé ci-dessous. On demande de préciser le code des méthodes `representant` et `estEgal`.

```
class MotCirculaire:
    """
    La classe MotCirculaire est une implémentation naïve des mots circulaires.
    On se contente ici de listes de caractères, que l'on implémente au moyen
    de chaînes de caractères (en concaténant la chaîne avec elle-même).
    """

    def __init__(self, representant):
        """
        Initialisation de la liste circulaire à partir d'une chaine de caractères.
        """
        self.chaine = representant*2
        self.longueur = len(representant)

    def __len__(self):
        """
        Renvoie la longueur de la liste circulaire.
        """
        return self.longueur

    def representant(self):
        """
        Renvoie le représentant initial du mot circulaire
        """

    def estEgal(self, autreMot):
        """
        Renvoie True si autreMot est un représentant du même mot circulaire.
        """
```

On remarquera que l'attribut `chaine` permet de tester l'appartenance d'un mot en considérant la chaîne `representant` concaténée à elle-même.

5. Donner une suite d'instructions créant deux instances de la classe `MotCirculaire` à partir des chaînes `01001` et `10010` puis permettant de vérifier au moyen de la méthode `estEgal` qu'elles représentent le même mot circulaire.

6. On dit qu'un mot circulaire $m = (m_0 \dots m_{p-1})$ est **un mot de De Bruijn** d'ordre (n, k) lorsque chaque mot de $M(n, k)$ apparaît exactement une fois dans $m_0 \dots m_{p+n-2}$ (où les indices sont considérés modulo p).

Par exemple, (0110) est un mot de De Bruijn d'ordre $(2, 2)$. En effet, les mots binaires de longueur 2 : 00, 01, 10 et 11 sont présents exactement une fois dans 01100.

Montrer que (002212011) est un mot de De Bruijn d'ordre $(2, 3)$.

7. Déterminer un mot de De Bruijn d'ordre $(3, 2)$.

8. Montrer que la longueur de tout représentant d'un mot de De Bruijn d'ordre (n, k) est inférieure ou égale à $n \times k^n$.

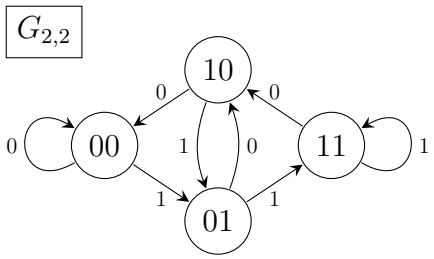
Partie C

Nous noterons $\boldsymbol{G_{n,k}}$ le graphe orienté et étiqueté dont les sommets sont les éléments de $M(n, k)$ et dont les arêtes sont définies comme suit : si u et v sont deux mots de n lettres sur l'alphabet $[\![0, k-1]\!]$ alors (u, v) est une arête de $G_{n,k}$ si

$$\exists a, b \in [\![0, k-1]\!],\ \exists x \in M(n-1, k) \text{ tel que } u = ax \text{ et } v = xb.$$

L'arête (u, v) aura alors l'étiquette b. Ainsi, le graphe $G_{4,2}$ admet $(0110, 1101)$ comme arête avec $a = 0$, $x = 110$ et $b = 1$.

La figure suivante donne une représentation du graphe $G_{2,2}$:



1. Représenter le graphe $G_{3,2}$.

2. La bibliothèque Python `networkx` peut être utilisée pour modéliser des graphes. Elle fournit une classe `DiGraph` permettant de définir un graphe orienté, avec pour méthodes :
- `add_edge` pour ajouter une arête et ses sommets,
- `nodes` pour retrouver les sommets,
- `edges` pour retrouver les arêtes.

Voici un exemple en console :

```
>>> from networkx import *       # chargement du module networkx
>>> G = DiGraph()                # définit G comme une instance de la classe DiGraph
>>> G.add_edge(1,2,label='a') # ajoute l'arête (1,2) (les sommets sont créés si
    besoin)
>>>                              # et donne 'a' pour étiquette à cette arête
>>> G[1][2]['label']             # retourne l'étiquette de l'arête (1,2)
'a'
>>> G.add_edge(1,'z')            # une autre arête
>>> G.nodes()                    # retourne la liste des sommets de G
[1, 2, 'z']
>>> G.edges()                    # retourne la liste des arêtes de G
[(1, 2), (1, 'z')]
```

Écrire une fonction `genGrapheDeBruijn` générant le graphe $G_{n,k}$. La fonction prendra en paramètres n et k et renverra une instance du graphe $G_{n,k}$. Notons qu'en `Python`, il est possible de retourner une instance G de la classe `DiGraph` avec l'instruction `return` G.

On rappelle que dans un graphe orienté G, **un circuit eulérien** est un chemin fermé passant une fois et une seule par chaque arête de G. Un graphe orienté G possédant un circuit eulérien est appelé **graphe eulérien**.

3. Montrer que $G_{3,2}$ est eulérien.

Dans la suite de ce problème, on admettra que $G_{n,k}$ **est eulérien**.

4. Trouver un circuit eulérien dans $G_{2,2}$ puis vérifier que la concaténation des étiquettes lues au fil de ce circuit donne un représentant d'un mot de De Bruijn d'ordre $(3, 2)$.

On admettra que la concaténation des étiquettes lues au fil d'un circuit eulérien de $G_{n,k}$ donne un représentant d'un mot de De Bruijn d'ordre $(n+1, k)$ et que les mots de De Bruijn d'ordre (n, k) ont tous la même longueur.

5. En déduire la longueur d'un mot de De Bruijn d'ordre (n, k).

6. La fonction `eulerian_circuit` du module `networkx` permet d'obtenir les arêtes d'un circuit eulérien (lorsqu'il en existe un) à partir d'un sommet donné. Ses paramètres sont l'instance de la classe `DiGraph`

dont on souhaite obtenir un circuit eulérien, ainsi que l'étiquette du sommet de départ. L'exemple suivant permet d'afficher les arcs du circuit eulérien d'un graphe ainsi que les étiquettes correspondantes.

```
>>> G = DiGraph()
>>> G.add_edge(0,1,label='a')
>>> G.add_edge(1,2,label='b')
>>> G.add_edge(2,0,label='c')
>>> liste1=[]
>>> liste2=[]
>>> for e in eulerian_circuit(G,0):
...     liste1.append(e)
...     liste2.append(G[e[0]][e[1]]['label'])
...
>>> liste1
[(0,1),(1,2),(2,0)]
>>> liste2
['a','b','c']
```

Écrire une fonction `genMotDeBruijn` qui prendra comme paramètres deux entiers n et k et renverra un représentant d'un mot de De Bruijn d'ordre (n, k) sous la forme d'une chaîne de caractères.

On suppose désormais qu'une classe `MotDeBruijn` a été écrite, ayant pour but de créer à partir de deux entiers n et k un objet représentant un mot de De Bruijn d'ordre (n, k). Cette classe hérite de la classe `MotCirculaire` et a deux attributs qui sont `n` et `k`. Elle contient trois méthodes sans paramètres `motn`, `genGrapheDeBruijn` et `genMotDeBruijn` adaptées des fonctions du même nom précédemment définies.

7. Écrire une fonction `estDeBruijn` permettant de déterminer si une chaîne définit un représentant pour un mot de De Bruijn d'ordre (n, k). La fonction prendra comme paramètres une chaîne de caractères, n et k. Elle renverra un booléen (`True` si la chaîne est un représentant). Cette fonction utilisera le résultat de la question C-5 et testera la présence de toutes les sous-chaînes.

Partie D

Dans cette partie, nous considérons une porte s'ouvrant avec un digicode à 4 chiffres. Il s'agit de trouver un nombre quelconque à 4 chiffres avec les touches 0, 1, 2, 3, 4, 5, 6, 7, 8 et 9.

1. Combien y a-t-il de combinaisons possibles ?

Il n'y a pas de validation : lorsque le code est composé, la porte s'ouvre, peu importe ce qui est tapé avant ou après. Par exemple si le code est 1256 la séquence 34**1256**98 ouvre la porte.

2. Une première méthode « naïve » est de tester tous les codes indépendamment les uns des autres. Combien de frappes de touches doit-on effectuer au maximum dans ce cas ?

3. Comment utiliser les mots de De Bruijn pour ouvrir la porte plus rapidement ?

4. Comparer la taille maximale des mots à écrire avec les deux méthodes.

5. Sachant qu'il faut en moyenne une seconde pour appuyer sur 4 touches, combien faut-il de temps pour essayer toutes les combinaisons avec chacune des méthodes ? La réponse sera donnée en heures, minutes, secondes.

6. Écrire une fonction `genCode` qui génère aléatoirement un code à 4 chiffres. La fonction ne prendra pas de paramètre et renverra une chaîne de caractères de 4 chiffres.

La fonction `randint` du module `random` pourra être utilisée. Sa syntaxe est décrite en annexe.

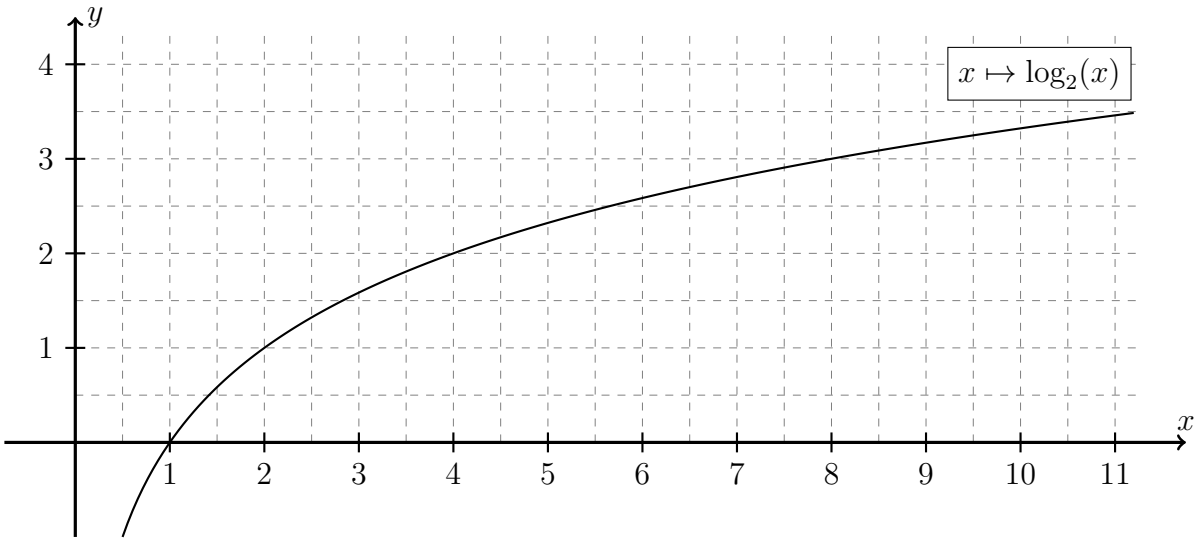
7. Écrire un programme qui :

– génère un code c aléatoirement,

– génère un mot m de De Bruijn d'ordre nécessaire à contenir c,
– retrouve le code c généré en parcourant le mot m, puis l'affiche,
– affiche le quotient $\frac{n_f}{n_{max}}$ où n_f représente le nombre de frappes nécessaires et n_{max} le nombre de frappes trouvé à la question 2.

Annexe

Courbe de la fonction logarithme de base 2



Langage Python

Listes

```
>>> maListe = [1,8,'e']    # définition d'un liste
>>> maListe[0]             # le premier élément d'une liste a l'indice 0
1
>>> maListe[1]
8
>>> maListe[-1]            # le dernier élément de la liste
'e'
>>> len(maListe)           # longueur d'une liste
3
>>> maListe.append(12)     # ajout d'un élément en fin de liste
>>> maListe
[1, 8, 'e', 12]
>>> maListe.remove(8)      # suppression du premier élément égale à 8
>>> maListe
[1, 'e', 12]
>>> maListe.pop(1)         # retourne l'élément d'indice 1 et le supprime de la liste
'e'
>>> maListe
[1, 12]
>>> maListe.insert(1,'a') # insert l'élément 'a' à l'indice 1 du tableau
>>> maListe
[1, 'a', 12]
```

Chaînes

```
>>> maChaine = 'Informatique'   # définition d'une chaîne de caractère
>>> len(maChaine)               # longueur d'une chaîne de caractère
12
>>> maChaine[0]                 # le premier caractère de la chaîne est d'indice 0
'I'
>>> maChaine += ' et mathematiques' # concaténation de chaînes
>>> maChaine
'Informatique et mathematiques'
>>> maChaine[-1]        # Un indice négatif permet un parcourt depuis la fin.
's'
>>> maChaine.count('e')       # le nombre de caractère 'e' dans la chaîne
4
```

Module `math`

```
>>> from math import *   # chargement du module math
>>> pi                   # le nombre pi
3.141592653589793
>>> cos(pi/3)            # cosinus en radian
0.5
>>> sin(pi/2)            # sinus en radian
1.0
>>> sqrt(2)              # racine de 2
1.4142135623730951
```

Module `random`

```
>>> from random import *   # chargement du module random
>>> random()               # renvoie aléatoirement un nombre entre 0 et 1 selon une
   loi uniforme.
0.5971130745072283
>>> randint(2,8)           # renvoie avec équiprobabilité un entier de l'intervalle [
   a,b]
3
>>> choice([1,2,4,'e',7])  # renvoie aléatoirement un élément de la liste
2
```