Problème 1. Mots et graphes de de Bruijn

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

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 désignera le logarithme de base 2, c'est-à-dire $\forall x > 0$, $\log(x) = \frac{\ln(x)}{\ln(2)}$ où ln est la fonction logarithme népérien.

Question 1.1. Questions préliminaires

(a) Énumérer les mots binaires de longueur 3.

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 $\mathscr{A}^n$. 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 $[\![0,k-1]\!]$.

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

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

(d) 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.*

1.1 Mots de de Bruijn

Question 1.2. Un **mot circulaire** est une séquence $(a_1a_2\cdots a_n)$ de lettres données avec un ordre circulaire, c'est-à-dire que la lettre a_1 suit a_n. Le mot $a_1\cdots a_n$ est un **représentant** du mot circulaire $(a_1\cdots a_n)$. Les mots $a_2\cdots a_na_1$, $a_3\cdots a_1a_2$, ..., $a_na_1\cdots a_{n-1}$ sont les autres représentants du même mot circulaire. 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 :

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

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

(b) Déterminer le nombre maximal de représentants d'un mot circulaire de n lettres. *La réponse devra être justifiée.*

(c) 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.*

Question 1.3. Une classe pour les mots circulaires 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.

```
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 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.

(a) Compléter le code des méthodes `representant` et `estEgal`.

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

Question 1.4. Mot de de Bruijn 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.

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

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

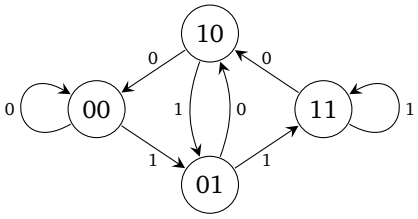
(c) Montrer que la longueur de tout représentant d'un mot de de Bruijn d'ordre (n,k) est inférieure ou égale à $n \times k^n$.

1.2 Graphes de de Bruijn

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 $G_{2,2}$:



Question 1.5. Représentation

(a) Représenter le graphe $G_{3,2}$.

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()                # création de G comme instance de DiGraph()
>>> G.add_edge(1,2,'label'='a') # ajoute l'arête (1,2) avec étiquette 'a'
                                 # (les sommets sont créés si besoin)
>>> 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')]
```

(b) 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`.

Question 1.6. 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**.

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

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

(b) 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.

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

Question 1.7. Génération de mot de de Bruijn 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']
```

(a) 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 trois attributs qui sont `n`, `k` et `representant`. Elle contient trois méthodes sans paramètres `motn`, `genGrapheDeBruijn` et `genMotDeBruijn` adaptées des fonctions du même nom précédemment définies. L'attribut `representant` est initialisé à l'aide de la méthode `genMotDeBruijn`.

(b) 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 1.6 (c) et testera la présence de toutes les sous-chaînes.*

1.3 Une application

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.

Question 1.8. Ouverture du digicode

(a) 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.

(b) 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 ?

(c) Comment utiliser les mots de de Bruijn pour ouvrir la porte plus rapidement ?

(d) Comparer la taille maximale des mots à écrire avec les deux méthodes.

(e) 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.

Question 1.9. Statistiques

(a) 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 :* `randint(a,b)` *renvoie un nombre entier aléatoire de l'intervalle* $[\![a, b]\!]$.

(b) Écrire un programme qui :

- génère un code c aléatoirement,
- génère un mot m de de Bruijn,
- 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 1.8 (b).