Algorithmes et Pensée Computationnelle

Probabilistic Algorithms

Le but de cette séance est de comprendre les algorithmes probabilistes. Ceux-ci permettent de résoudre des problèmes complexes de en relativement peu de temps. La contrepartie est que le résultat obtenu est généralement une solution approximée du problème initial. Ils demeurent néanmoins très utile pour beaucoup d'application.

1 Monte-Carlo

Question 1: (*10 minutes*) **Un jeu de hasard : Python**

Supposez que vous lanciez une pièce de monnaie l fois et que vous voulez calculez la probabilité d'avoir un certains nombre de pile. Vous devez programmer un algorithme probabiliste, permettant de calculer cette probabilité. Pour ce faire, vous devez compléter la fonction *proba(n,l,iter)* contenue dans le fichier Piece.py. La fonction *Piece(l)* permet de créer une liste contenant des 0 et des 1 aléatoirement avec une probabilité $\frac{1}{2}$. Considérez un chiffre 1 comme une réussite (pile) et 0 comme un échec (face).

Conseil

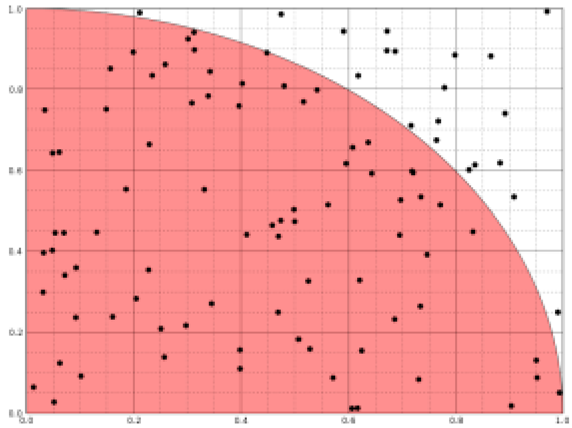
Pour estimez empiriquement la probabilité d'un événement, comptez le nombre de fois que l'événement en question se produit en effectuant un nombre d'essai. Puis divisez le nombre d'occurence de l'événement par le nombre total d'essai. Par exemple, si vous vous voulez estimer la probabilité d'obtenir un 2 avec un dé. Lancez le dé 1000 fois, comptez le nombre de fois que vous obtenez 2, et divisez le résultat par 1000.

Solution

```python

import numpy as np

#La fonction Piece retourn une liste contenant des 0 et des 1, considérez un 1 comme un succès, i.e. une fois ou la pièce tombe sur pile, et 0 comme un échec
def Piece(l):
    return np.random.randint(0,2,l)



def proba(n,l,iter):
    #Codez , n correspond au nombre de succès et l au nombre d'essaie. Iter correspond au nombre d'expérience que vous allez réaliser pour obtenir la réponse. Cela devrait être grand mais pas trop (sinon le programme prendra trop de temp.)
    #10000 est un bon nombre d'itération.
    proba = 0
    for i in range(iter):
        temp = Piece(l)#On simule une expérience de l lancé.
        count = temp.sum()#On compte le nombre de fois que l'on obtient pile
        if count == n:#Si le nombre de pile obtenue correspond à la probabilité que l'on veut estimer
            proba +=1#On ajoute 1 à notre estimateur de probabilité



    return proba/iter#Divise notre estimateur de probabilité par le nombre total d'expérience réalisée.


n = 5
l = 10
print(”La probabilité d'avoir {} pile en {} lancés de pièce est approximativement égale à {}”.format(n,l,proba(n,l,10000)))
```

Question 2: (🕒 *20 minutes*) **Une approximation de π : Python**

L'objectif de cet exercice est programmer un algorithme probabiliste permettant d'approximer le chiffre π. Imaginez un plan en sur lequel $0 < x < 1$ et $0 < y < 1$. Sur ce dernier, nous allons dessiner un quart de cercle centré en (0,0) et avec un rayon de 1. Par conséquent, un point dans cette espace se trouve à l'intérieur du cercle si $x^2 + y^2 < 1$. Vous trouverez ci-dessous un schéma de la situation :



La première étape de cette exercice consiste a créer une fonction permettant de déterminer si un point est à l'intérieur (zone rouge) ou a l'extérieur du cercle. Puis, générez 10000 points dans cette espace (x et y devrait appartenir à [0,1]). Pour ce faire, vous pouvez utliser la fonction *random.random()* après avoir importé le module **random**. Vous pouvez obtenir l'approximation de π à partir de la formule suivante : $\pi \approx [\frac{\text{Nombre de point dans le cercle}}{\text{Nombre de point total}}] \cdot 4$. Votre réponse devrait être assez proche du vrai chiffre π.

Conseil

La fonction random.random() génère aléatoirement un chiffre compris entre 0 et 1. Etant donné que vous devez simulez des points en 2 dimension, vous devrez utiliser 2 fois cette fonction.

Solution

```
import random

def inside(point):#Point définit sous la forme d'un tuple

    if (point[0]**2+point[1]**2) < 1:
        return 1

    else:
        return 0

def app():
    count = 0 #On initialise le nombre de point dans le cercle
    for i in range(10000):
        temp1 = random.random()#Génére la première coordonnée
        temp2 = random.random()#Génère la dexuième coordonnée
        temp = [temp1,temp2]#Crée le point

        count += inside(temp)#On appelle la fonction. Si le point est dans le cercle, elle retourn 1, par conséquent on ajoute 1 au compteur. Sinon elle retourne 0, on ajoute donc rien.

    return count/10000*4#Retourn selon la formule.

print("L'approximation du chiffre pi est : {}".format(app()))
```

Question 3: (🕒 *15 minutes*) **Un exemple simple de la chaîne de Markov**

Les slides nous ont donné un exemple de la chaîne de Markov. Celui-ci comprend une matrice de transition et un calcul des probabilités de 3 états $x = (1, 2, 3)$ (x est un vecteur ayant 3 éléments) à $t + 3$ i.e $x^{(t+3)}$.

Avant de continuer, assurez-vous que vous comprenez la formule de la probabilité conditionnelle dans les slides, et les opérations basiques de matrices et vecteurs qui y sont présentées (puissance et produit scalaire).

On va formaliser le concept de la chaîne de Markov avec les notations suivantes.

I, Un processus ou une séquence $X_0, X_1, X_2, ..., X_n$ $(0, 1, 2, ..., n$ signifiant de différents moments) est une chaîne de Markov si

$$P\Big(X_{n+1} = j|X_0 = i_0, X_1 = i_1, ..., X_{n-1} = i_{n-1}, X_n = i\Big) = P\left(X_{n+1} = j|X_n = i\right). \qquad (1)$$

En d'autres termes, toute information utile pour la prédiction du futur de la valeur X d'une chaîne de Markov est uniquement dans l'état présent.

II, Le nombre $P\left(X_{n+1} = j|X_n = i\right)$ est appelé *probabilité de transition* de l'état i à l'état j (en un pas), et on écrit :

$$p_{ij} = P\left(X_{n+1} = j|X_n = i\right) \qquad (2)$$

La matrice $\mathbf{P}$ dont l'élément à l'indice (i, j) (ligne i, colonne j) est p_{ij} est appelée *matrice de transition*. Si on a N états, P a dimension $N \times N$ (N lignes et N colonnes). Si les chaînes sont homogènes, $\mathbf{P}$ a deux propriétés importantes : i, $p_{ij} \geq 0$ et ii, $\sum_i p_{ij} = 1$ (vérifiez que c'est le cas dans l'exemple des slides).

III, Soit $\mu_n = (\mu_n(1), ..., \mu_n(N))$ un vecteur-ligne des probabilités, avec $\mu_n(i) = P(X_n = i)$. Par exemple, si on a 3 états et $\mu_2 = (0.5, 0.2, 0.3)$, on peut dire qu'à temps 2, la probabilité est 0.5 qu'on soit à l'état 1, 0.2 qu'on soit à l'état 2, et 0.3 qu'on soit à 3. **La variable $x^{(t+3)}$ des slides serait μ_3 avec ces notations** !

μ_n est aussi appelé probabilités marginales, qui indiquent les probabilités des états à temps n, et elles sont calculées comme suit (vérifiez que cette formule s'accorde avec le calcul de $x^{(t+3)}$ des slides) :

$$\mu_n = \mu_0 \mathbf{P}^n$$

μ_0 est donc appelé *la loi initiale* (la loi de X_0) ; dans les slides, $\mu_0 = (0,\ 1,\ 0)$. En général, on a qu'à connaître μ_0 et $\mathbf{P}$ pour simuler une chaîne de Markov. Cette information sera utile pour l'exercice 5.

Et on a ci-dessous un résumé de la terminologie :

1. $\mathbf{P}(i, j)$: élément à ligne i et colonne j de la matrice $\mathbf{P}$.
2. Matrice de transition $\mathbf{P}$ a $\mathbf{P}(i, j) = P(X_{n+1} = j|X_n = i) = p_{ij}$.
3. $\mathbf{P}_n = \mathbf{P}^n$.
4. Probabilité marginale : $\mu_n(i) = P(X_n = i)$.
5. $\mu_n = \mu_0 \mathbf{P}^n$

Etant donné que $X_0, X_1, ...$ est une chaîne de Markov avec 3 états $\{0, 1, 2\}$ et la matrice de transition :

$$\mathbf{P} = \begin{bmatrix} 0.1 & 0.2 & 0.7 \\ 0.9 & 0.1 & 0.0 \\ 0.1 & 0.8 & 0.1 \end{bmatrix}$$

Supposons que $\mu_0 = (0.3,\ 0.4,\ 0.3)$. Trouvez $P(X_0 = 0, X_1 = 1, X_2 = 2)$ et $P(X_0 = 0, X_1 = 1, X_2 = 1)$.

Conseil

Cet exercice vous demande de trouver deux probabilités **jointes**. Peut-être vous rappelez-vous qu'en général,

$$P(X = x, Y = y) = P(X = x)P(Y = y|X = x).$$

Pouvez-vous le reformuler avec 3 variables ? En plus, notez bien que $X_0, X_1, ...$ est une chaîne de Markov i.e $P\Big(X_{n+1} = j|X_0 = i_0, X_1 = i_1, ..., X_{n-1} = i_{n-1}, X_n = i\Big) = P\left(X_{n+1} = j|X_n = i\right)$.

Solution

$$P(X_0 = 0, X_1 = 1, X_2 = 2) = P(X_0 = 0) \times p_{01} \times p_{12} = 0.3 \times 0.1 \times 0.7 = 0.021 \quad (3)$$
$$P(X_0 = 0, X_1 = 1, X_2 = 1) = P(X_0 = 0) \times p_{01} \times p_{11} = 0.3 \times 0.1 \times 0.1 = 0.003 \quad (4)$$

Question 4: (*7 minutes*) **Probabilités marginales**
En utilisant la loi initiale μ_0 et la matrice $\mathbf{P}$ de Question 3, trouvez μ_1, μ_2.

Conseil

Relisez et familiarisez-vous avec les notations et la terminologie ci-dessus ! Si les slides s'avèrent plus utiles, considérez $\mu_1 = x^{(t+1)}$, $\mu_2 = x^{(t+2)}$.

On peut aussi essayer de les trouver en écrivant un programme Python ! Nous vous fournissons une fonction qui calcule le produit scalaire entre un vecteur et une matrice. Essayez d'écrire une fonction pour trouver la puissance d'une matrice (en utilisant la fonction de produit scalaire) et puis une autre fonction pour les probabilités marginales.

```python
def produit_scalaire(vec, mat):
    # vec: liste de n elements
    # mat: liste de n sous-listes

    result = []
    for i in range(len(mat[0])): #iterer sur les colonnes de la matrice
        total = 0
        for j in range(len(vec)): # iterer sur les elements du vecteur et les lignes de la matrice
            total += vec[j] * mat[j][i]
        result.append(total)

    return result

def puissance_mat(mat, n):
    # P: liste de listes
    # n: integer

    new_mat = mat
    for i in range(n-1):
        dot_prod = [] # produit scalaire entre chaque ligne et la matrice complete
        ...

    return new_mat

def prob_marginales(mu_0, P, n):

    return ...


mu_0 = [0.3, 0.4, 0.3]
P = [[0.1, 0.2, 0.7],
    [0.9, 0.1, 0.],
    [0.1, 0.8, 0.1]]
n = 2 # puissance

print(prob_marginales(mu_0, P, n))
```

Conseil

Souvenez-vous que $\mathbf{P}^2$ est le produit scalaire entre $\mathbf{P}$ et $\mathbf{P}$ soi-même ! Chaque ligne de $\mathbf{P}^2$ sera donc le produit scalaire entre une ligne de $\mathbf{P}$ et $\mathbf{P}$.

>_ Solution

$$\mu_1 = \mu_0 \mathbf{P}^1 = \begin{pmatrix} 0.42 & 0.34 & 0.24 \end{pmatrix} \tag{5}$$

$$\mu_2 = \mu_0 \mathbf{P}^2 = \begin{pmatrix} 0.37 & 0.31 & 0.32 \end{pmatrix} \tag{6}$$

```
def produit_scalaire(vec, mat):
    # vec: liste de n elements
    # mat: liste de n sous-listes

    result = []
    for i in range(len(mat[0])): #iterer sur les colonnes de la matrice
        total = 0
        for j in range(len(vec)): # iterer sur les elements du vecteur et les lignes de la matrice
            total += vec[j] * mat[j][i]
        result.append(total)

    return result

def puissance_mat(mat, n):
    # P: liste de listes
    # n: integer

    new_mat = mat
    for i in range(n-1):
        dot_prod = []
        for ligne in mat:
            dot_prod.append(produit_scalaire(ligne, new_mat))
        new_mat = dot_prod
    return new_mat

def prob_marginales(mu_0, P, n):

    return produit_scalaire(mu_0, puissance_mat(P, n))


mu_0 = [0.3, 0.4, 0.3]
P = [[0.1, 0.2, 0.7],
    [0.9, 0.1, 0.],
    [0.1, 0.8, 0.1]]
n = 2 # puissance

print(prob_marginales(mu_0, P, n))
```

Question 5: (*20 minutes*) **Simuler une chaîne de Markov**

Pour cet exercice, vous n'avez pas à utiliser les calculs dans l'exercise 3 et 4.

Comme déjà mentionné plus haut, la simulation d'une chaîne de Markov $(X_0, X_1, ...)$ exige seulement deux éléments : la loi initiale μ_0 et la matrice de transition $\mathbf{P}$. Spécifiquement, l'algorithme est :

1. Supposer que les probabilités initiales (les probabilités des états potentiels de X_0) sont dans le vecteur μ_0. Trouver X_0.
2. Le résultat de l'étape 1 est donc $X_0 = i$, l'état de X à temps 0 ; obtenir X_1 selon les probabilités à la ith ligne de $\mathbf{P}$ où $P(X_1 = j | X_0 = i) = p_{ij}$. Trouver X_1.
3. Le résultat de l'étape 2 est $X_1 = j$, l'état de X à temps 1 ; obtenir $X_2 \sim \mathbf{P}$ où $P(X_2 = k | X_1 = j) = p_{jk}$.
4. Répéter jusqu'à la fin (nombre d'iterations est arbitraire).

Ecrivez une fonction simple afin d'implémenter l'algorithme ci-dessus. Nommez-la **sim_markov()**, celle-ci prend comme parametres **P**, **mu_0** et **n_iters**. Essayez avec des valeurs différentes de **P**, **mu_0** et **n_iters**.

```
def sim_markov(mu_0, P, n_iters=500):
    # mu_0 (n,1) # probabilités initiales - n états
    # P (n, n) # matrice de transition

```

```python
    states = range(len(mu_0)) # e.g si la longueur de mu_0 est 2, on aura 2 états 0, 1
    X0 = ...
    ...

P = [[0.1, 0.9],
    [0.7, 0.3]]
mu_0 = [0.3, 0.7]
print(sim_markov(mu_0, P))
```

Conseil

La méthode **random.choices()** s'avéra utile !

Solution

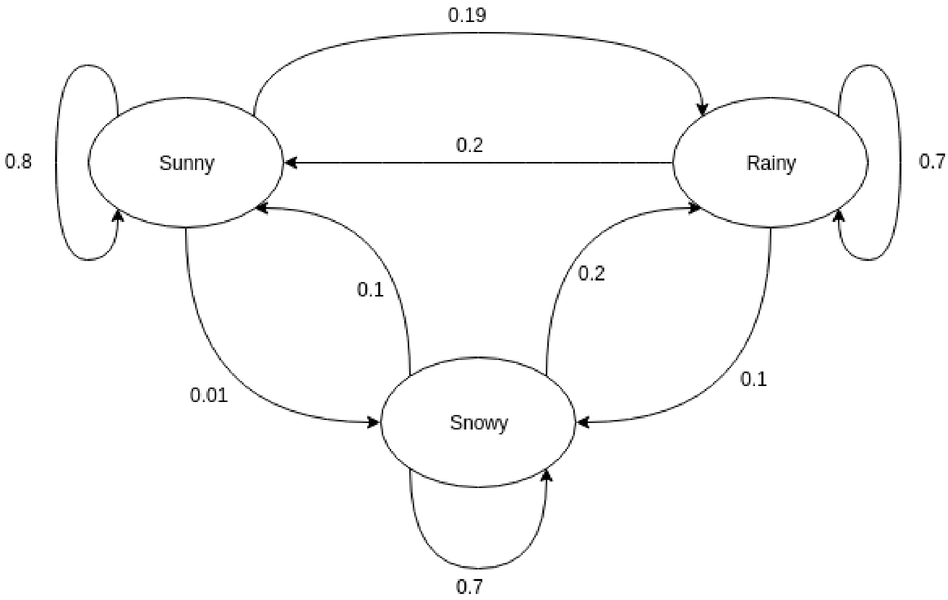
```python
import random

def sim_markov(mu_0, P, n_iters=500):
    # mu_0 (n,1) # probabilités initiales – n états
    # P (n, n) # matrice de transition

    states = range(len(mu_0)) # e.g si la longueur de mu_0 est 2, on aura 2 états 0, 1
    X0 = random.choices(states, mu_0)[0]

    future_states = []
    future_states.append(X0)
    for i in range(n_iters):
        next_state = random.choices(states, P[future_states[i]])[0]
        future_states.append(next_state)

    return future_states

P = [[0.1, 0.9],
    [0.7, 0.3]]

mu_0 = [0.3, 0.7]

print(sim_mc(mu_0, P))
```

Question 6: (*20 minutes*) **Coder une chaîne de Markov avec le dictionnaire Python**

Cette fois-ci, vous allez utiliser un dictionnaire au lieu de vecteurs et matrices !

Supposez qu'il y a trois choix d'états avec les transitions dans l'image ci-dessous. Supposez également que la loi initiale des états sont (0.3, 0.2, 0.5) pour Sunny, Snowy, et Rainy, respectivement. Simulez une chaîne de Markov en utilisant un dictionnaire imbriqué, écrit comme suit

```python
prob_transition = {
    'Sunny': {'Sunny': 0.8, 'Rainy': 0.19,  'Snowy': 0.01},
    'Rainy': {'Sunny': 0.2, 'Rainy': 0.7, 'Snowy': 0.1},
    'Snowy': {'Sunny': 0.1, 'Rainy': 0.2, 'Snowy': 0.7}
}
```



Complétez le programme ci-dessous.

```python
import random


def sim_markov_dict(mu_0, P, n_iters):

    # liste d'etats
    states = list(mu_0.keys())

    # premier etat
    current_state = ...

    future_states = []
    ...

    return ...

mu_0 = {'Sunny': 0.3, 'Snowy': 0.2, 'Rainy': 0.5}
prob_transition = {
    'Sunny': {'Sunny': 0.8, 'Rainy': 0.19,  'Snowy': 0.01},
    'Rainy': {'Sunny': 0.2, 'Rainy': 0.7, 'Snowy': 0.1},
    'Snowy': {'Sunny': 0.1, 'Rainy': 0.2, 'Snowy': 0.7}
}

sim_markov_dict(mu_0, prob_transition, 50)
```

Conseil

random.choices() sera de nouveau pertinente !

Solution

```python
import random


def sim_markov_dict(mu_0, P, n_iters):

    # liste d'etats
    states = list(mu_0.keys())

    # premier etat
    current_state = random.choices(states, list(mu_0.values()))[0]

    future_states = []
    for i in range(n_iters):
        next_probs = list(P[current_state].values())
        next_state = random.choices(states, next_probs)[0]
        future_states.append(next_state)
        current_state = next_state
    return future_states

mu_0 = {'Sunny': 0.3, 'Snowy': 0.2, 'Rainy': 0.5}
prob_transition = {
    'Sunny': {'Sunny': 0.8, 'Rainy': 0.19,  'Snowy': 0.01},
    'Rainy': {'Sunny': 0.2, 'Rainy': 0.7, 'Snowy': 0.1},
    'Snowy': {'Sunny': 0.1, 'Rainy': 0.2, 'Snowy': 0.7}
}

sim_markov_dict(mu_0, prob_transition, 50)
```

Question 7: (*20 minutes*) **Insertion dans une Treap** Une Treap est un arbre binaire où chaque sommet v a 2 valeurs, une clé **v.key** et une priorité **v.priority**. Une treap estimer un arbre de recherche binaire en ce qui concerne les valeurs clés et une heap en ce qui concerne les valeurs prioritaires.

Dans cet exercice, vous allez implémenter une fonction pour insérer un noeud dans une treap. Vous avez le squellette de code suivant à remplir.

```python
from random import randrange

# Un noeud de la treap
class TreapNode:
    def __init__(self, data, priority=100, left=None, right=None):
        self.data = data
        self.priority = randrange(priority)
        self.left = left
        self.right = right

# Fonction pour faire une rotation à gauche
def rotateLeft(root):

    R = root.right
    X = root.right.left

    # rotate
    R.left = root
    root.right = X

    # set root
    return R


# Fonction pour faire une rotation à droite
def rotateRight(root):

    L = root.left
    Y = root.left.right

    # rotation
    L.right = root
```

```
        root.left = Y

        # retourne la nouvelle racine
        return L


# Fonction récursive pour insérer une clé avec une priorité dans une Treap
def insertNode(root, data):
    # TODO
        return root


# Affiche les noeuds de la treap
def printTreap(root, space):
        height = 10

        if root is None:
                return

        space += height
        printTreap(root.right, space)

        for i in range(height, space):
                print(' ', end='')

        print((root.data, root.priority))
        printTreap(root.left, space)


if __name__ == '__main__':
        # Clés de la treap
        keys = [5, 2, 1, 4, 9, 8, 10]

        # Construction de la treap
        root = None
        for key in keys:
                root = insertNode(root, key)

        printTreap(root, 0)
```

Conseil

La fonction **insertNode** est récursive. Inspirez-vous de l'insertion dans un arbre de recherche binaire, mais n'oubliez de vérifier que la propriété de la heap est satisfaite après avoir inséré.
La propriété de la heap à satisfaire est que la priorité de la racine doit toujours être plus grande que celle de ses noeuds enfants. **rotateLeft** et **rotateRight** permettent de réarranger les noeuds de façon à ce que la propriété de la heap soit satisfaite. Vous pouvez vous réferer à l'illustration ci-dessous pour avoir une idée de comment fonctionne les rotations.

```
/* T1, T2 and T3 are subtrees of the tree rooted with y
  (on left side) or x (on right side)
                y                                x
               / \     Right Rotation          /   \
              x   T3   - - - - - - - >        T1    y
             / \       < - - - - - - -             / \
            T1  T2     Left Rotation              T2  T3 */
```

Solution

```python
from random import randrange

# Un noeud de la treap
class TreapNode:
    def __init__(self, data, priority=100, left=None, right=None):
        self.data = data
        self.priority = randrange(priority)
        self.left = left
        self.right = right

# Fonction pour faire une rotation à gauche
def rotateLeft(root):

    R = root.right
    X = root.right.left

    # rotate
    R.left = root
    root.right = X

    # set root
    return R


# Fonction pour faire une rotation à droite
def rotateRight(root):

    L = root.left
    Y = root.left.right

    # rotation
    L.right = root
    root.left = Y

    # retourne la nouvelle racine
    return L
```

Solution

```python
# Fonction récursive pour insérer une clé avec une priorité dans une Treap
def insertNode(root, data):

    if root is None:
        return TreapNode(data)

    # si data est inférieure à celle la racine root, insérer dans le sous−abre gauche
    # sinon insérer dans le sous−arbre droit
    if data < root.data:
        root.left = insertNode(root.left, data)

        # faire une rotation à droite si la propriété de la heap est violée
        if root.left and root.left.priority > root.priority:
            root = rotateRight(root)
    else:
        root.right = insertNode(root.right, data)

        # faire une rotation à gauche si la propriété de la heap est violée
        if root.right and root.right.priority > root.priority:
            root = rotateLeft(root)

    return root


# Affiche les noeuds de la treap
def printTreap(root, space):
    height = 10

    if root is None:
        return

    space += height
    printTreap(root.right, space)

    for i in range(height, space):
        print(' ', end='')

    print((root.data, root.priority))
    printTreap(root.left, space)


if __name__ == '__main__':
    # Clés de la treap
    keys = [5, 2, 1, 4, 9, 8, 10]

    # Construction de la treap
    root = None
    for key in keys:
        root = insertNode(root, key)

    printTreap(root, 0)
```

Question 8: (*20 minutes*) **Fingerprinting : Une mission pour l'agente secrète Alice** Dans cet exercice, vous prendrez le rôle de l'agente secrète Alice. Cette dernière enquêtait sur la disparition de son collègue, l'agent Bob, et se doutait que l'indice clé qui la ménera à la vérité se trouvait dans la boîte mail de Bob. Alice arriva à trouver un bout de papier avec écrit dessus : "Mon mot de passe est l'empreinte de ceciestmonmotdepasse". Aidez Alice à trouver l'empreinte du mot de passe !

Pour cela, vous devez compléter deux fonctions :

1. **is_a_prime_number(num)** qui vérifie que **num** est un nombre premier ou pas. Un nombre premier est un entier naturel qui admet exactement deux diviseurs distincts entiers et positifs. Ces deux diviseurs sont 1 et le nombre considéré, puisque tout nombre a pour diviseurs 1 et lui-même, les nombres premiers étant ceux qui n'en possèdent aucun autre.
2. **fingerprinting(p, message)** qui implémente l'algorithme de fingerprinting suivant :

(a) Si **p** est un nombre premier, calculez la valeur de hachage de la chaîne à l'aide de la fonction **hash(...)**, puis calculez le modulo du résultat du hachage.

(b) Sinon, imprimez un message qui dit que le nombre n'est pas un nombre premier.

Si vous réussissez à implémenter les deux fonctions correctement, le code vous imprimera : **Connection réussie? True**.

À vos ordis, détectives !

```
import base64

# num est un nombre entier
def is_a_prime_number(num):
    #TODO


# p est un nombre premier et message est une chaine de caractères
def fingerprinting(p, message):
    #TODO


# password est une chaine de caractères et your details est un tuple avec le
# format suivant (nombre premier, hash du mot de passe)
def login(password, your_details):
    return your_details[1] % your_details[0] == fingerprinting(your_details[0], password)

if __name__ == "__main__":
    password = "ceciestmonmotdepasse"
    your_details = (19, hash(password))
    success = login(password, your_details)

    print("Connection réussie? " + str(success))
    if success:
        message = '''SmUgc2VyYWlzIGNvbmZpbsOpIGNoZXogbWVzIHBhcmVudHMgw
                6AgbGEgY2FtcGFnbmUgbGVzIGRldXggcHJvY2hhaW5lIHNlbWF
                pbmVzLCBldCBqZSBuJ2F1cmFpcyBwYXMgYWNjw6hzIMOgIEludGV
                ybmV0LiDDgCBiaWVudMO0dCE='''
        print(base64.b64decode(message).decode())
```

>_ Solution

```python
import base64

# num est un nombre entier
def is_a_prime_number(num):
    if num <= 1:
        return False
    for i in range(2, int(num**.5)):
        if num % i == 0:
            return False
    return True

# p est un nombre premier et message est une chaine de caractères
def fingerprinting(p, message):
    if is_a_prime_number(p):
        result = hash(message) % p
        return result
    print(str(p) + " is not a prime number!")

# password est une chaine de caractères et your details est un tuple avec le
# format suivant (nombre premier, hash du mot de passe)
def login(password, your_details):
    return your_details[1] % your_details[0] == fingerprinting(your_details[0], password)

if __name__ == "__main__":
    password = "ceciestmonmotdepasse"
    your_details = (19, hash(password))
    success = login(password, your_details)

    print("Connection réussie? " + str(success))
    if success:
        message = '''SmUgc2VyYWlzIGNvbmZpbsOpIGNoZXogbWVzIHBhcmVudHMgw
                6AgbGEgY2FtcGFnbmUgbGVzIGRldXggcHJvY2hhaW5lIHNlbWF
                pbmVzLCBldCBqZSBuJ2F1cmFpcyBwYXMgYWNjw6hzIMOgIEludGV
                ybmV0LiDDgCBiaWVudMO0dCE='''
        print(base64.b64decode(message).decode())
```