Algorithmes et Pensée Computationnelle

Algorithmes probabilistes

Le but de cette séance est de comprendre les algorithmes probabilistes. Ceux-ci permettent de résoudre des problèmes complexes en relativement peu de temps. La contrepartie est que le résultat obtenu est généralement une solution approximative du problème initial. Néanmoins, ces algorithmes demeurent très utiles pour beaucoup d'applications.

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 certain nombre de piles. 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** (Dans le dossier **Ressources**). 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

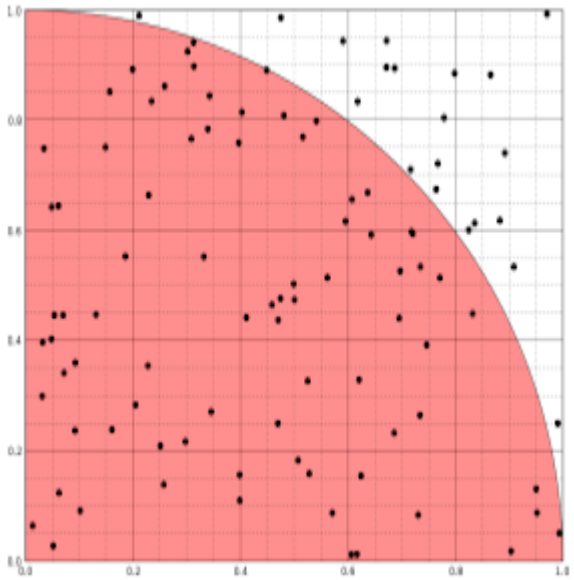
Pour estimer empiriquement la probabilité d'un événement, comptez le nombre de fois que l'événement en question se produit en effectuant un nombre d'essai. Puis divisez le nombre d'occurrence de l'événement par le nombre total d'essai. Par exemple, si vous voulez estimer la probabilité d'obtenir un 2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

Solution

```
1  import random
2
3
4  # La fonction Piece retourne une liste contenant des 0 et des 1, considérez un 1 comme un succès, i.e. une fois où
       la pièce tombe sur pile, et 0 comme un échec
5  def Piece(l):
6      return [random.randint(0, 1) for i in range(l)]
7
8
9  def proba(n, l, iter=10000):
10     # n correspond au nombre de succès et l au nombre d'essais. Iter correspond au nombre d'expérience que vous
           allez
11     # réaliser pour obtenir la réponse. Cela devrait être grand mais pas trop (sinon le programme prendra trop de
12     # temps à s'exécuter). 10000 est un bon nombre d'itérations.
13     proba = 0
14     for i in range(iter):
15         temp = Piece(l)  # On simule une expérience de l lancés.
16         count = sum(temp)  # On compte le nombre de fois que l'on obtient pile
17         if count == n:  # Si le nombre de pile obtenu correspond à la probabilité que l'on veut estimer
18             proba += 1  # On ajoute 1 à notre estimateur de probabilité
19     return proba / iter  # Divise notre estimateur de probabilité par le nombre total d'expériences réalisées.
20
21
22 n = 5
23 l = 10
24 print("La probabilité d'avoir {} pile en {} lancés de pièce est approximativement égale à {}".format(n, l,
       proba(n, l,
25                                                                                    10000)))
```

Question 2: (🕓 *20 minutes*) **Une approximation de π : Python**
L'objectif de cet exercice est de programmer un algorithme probabiliste permettant d'approximer le chiffre π. Imaginez un plan 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 cet espace se trouve à l'intérieur du cercle si $x^2 + y^2 < 1$. Vous trouverez ci-dessous une illustration de la situation :



La première étape de cet exercice consiste à créer une fonction permettant de déterminer si un point est à l'intérieur (zone rouge) ou à l'extérieur du cercle. Puis, générez 10000 points dans cet espace (x et y devrait appartenir à l'intervalle [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 : $\pi \approx [\frac{\text{Nombre de points dans le cercle}}{\text{Nombre total de points}}] \cdot 4$. 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 simuler des points en 2 dimensions, vous devrez utiliser 2 fois cette fonction.

Solution

```
1  import random
2
3  def inside(point):#Point définit sous la forme d'un tuple
4      # Cette fonction permet de vérifier si un point se trouve à l'intérieur du cercle
5      if (point[0]**2+point[1]**2) < 1:
6          return 1
7
8      else:
9          return 0
10
11 def app():
12     count = 0 #On initialise le nombre de points dans le cercle
13     for i in range(10000):
14         temp1 = random.random()#Génére la première coordonnée
15         temp2 = random.random()#Génère la deuxième coordonnée
16         temp = [temp1,temp2]#Crée le point
17
18         count += inside(temp)#On appelle la fonction. Si le point est dans le cercle, elle retourne 1, par conséquent
             on ajoute 1 au compteur. Sinon elle retourne 0, on ajoute donc rien.
19
20     return count/10000*4#Retourne selon la formule donnée dans l'exercice.
21
22 print("L'approximation du chiffre pi est : {}".format(app()))
```

2 Fingerprinting

Question 3: (*20 minutes*) **Fingerprinting : Une mission pour l'agente secrète Alice : Python**
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 mènerait 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 !

```python
import base64

# num est un nombre entier
def is_a_prime_number(num):
    # Partie à compléter


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


# 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)



# Début de votre programme
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())
```

3 Las Vegas

Question 4: (*10 minutes*) **Quicksort - Algorithme de Las Vegas : Python**
Un algorithme de Las Vegas est un algorithme probabiliste qui a la particularité de toujours trouver le résultat correct lorsqu'il existe. Son inconvénient est que sa complexité temporelle ne peut être garantie à l'avance car elle dépend des données passées en paramètres.
Dans cet exercice, vous allez implémenter un algorithme de tri rapide (quicksort) sur une liste d'éléments.

L'algorithme de tri rapide applique un paradigme *divide-and-conquer* afin de trier un ensemble de nombres **A**. Il fonctionne en trois étapes :

1. il choisit d'abord un élément pivot, **A[q]**, en utilisant un générateur de nombres aléatoires (d'où sa nature d'algorithme dit probabiliste) ;
2. puis il réorganise le tableau en deux sous-tableaux $A[p...q-1]$ et $A[q+1...r]$, où les éléments des premier et deuxième tableaux sont respectivement plus petits et plus grands que $A[q]$.
3. L'algorithme applique ensuite récursivement les étapes de tri rapide ci-dessus sur les deux tableaux indépendants, produisant ainsi un tableau entièrement trié.

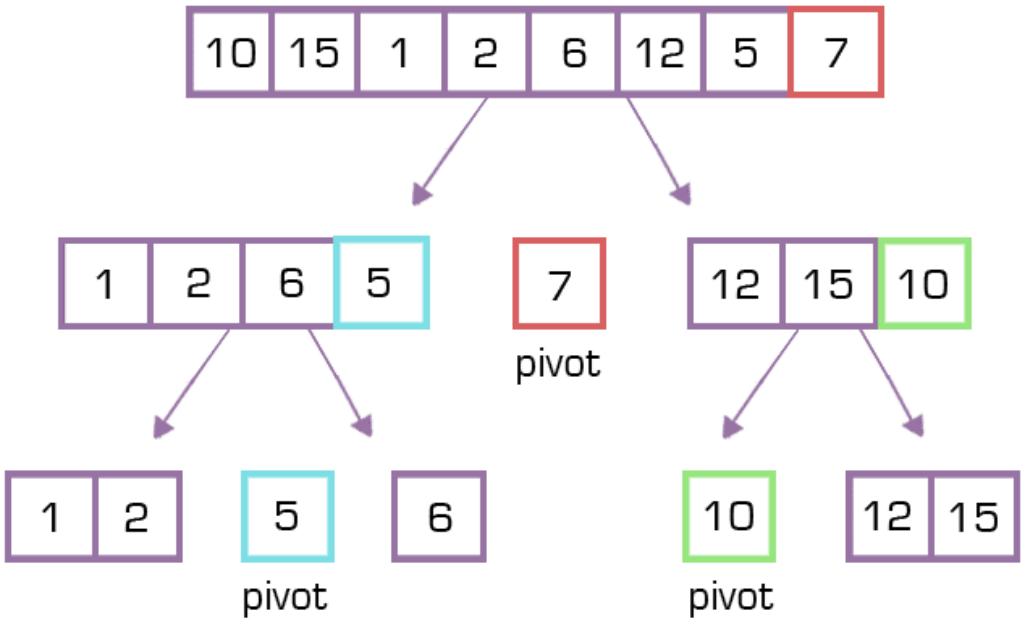


FIGURE 1 – Illustration de l'algorithme de tri rapide

Complétez le code suivant :

```
1  import random
2
3  # lst représente la liste à trier, l l'index 0 et r la taille de la liste –1
4  def sort(lst, l, r):
5      # mettre une condition pour arrêter la récursivité
6
7
8      pivot_index = ... # Partie à compléter: Choisissez un pivot compris entre 0 et la longueur de votre liste – 1
9
10     # Déplacer votre pivot dans votre liste
11
12     # Partitionnez votre liste de telle sorte que les éléments plus petits que le pivot soient placés avant celui–ci et les é
         léments plus grands soient placés après
13
14     # Replacer votre pivot à l'endroit adéquat
15
16     # Effectuez le tri de fa con récursive sur les parties gauches et droites de la liste
```

```
17
18  def quicksort(items):
19      if items is None or len(items) < 2:
20          return
21      sort(items, 0, len(items) - 1)
22
23  l = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
24  quicksort(l)
25  print('Liste triée:   ', l)
```

Conseil

Pour le choix de votre élément pivot, pensez à utiliser la méthode **randint**() de la librairie **random**.

Solution

```
1   import random
2
3   # lst représente la liste à trier, l l'index 0 et r la taille de la liste -1
4   def sort(lst, l, r):
5       # Dans le meilleur des cas, on arrête la récursivité
6       if r <= l:
7           return
8
9       # Choix du pivot
10      pivot_index = random.randint(l, r)
11
12      # On déplace le pivot au premier élément
13      lst[l], lst[pivot_index] = lst[pivot_index], lst[l]
14
15      # partition
16      i = l
17      for j in range(l+1, r+1):
18          if lst[j] < lst[l]:
19              i += 1
20              lst[i], lst[j] = lst[j], lst[i]
21
22      # On place le pivot à la position adéquate
23      lst[i], lst[l] = lst[l], lst[i]
24
25      # On effectue le tri de fa con récursive sur les parties gauches et droites de la liste
26      sort(lst, l, i-1)
27      sort(lst, i+1, r)
28
29  def quicksort(items):
30      if items is None or len(items) < 2:
31          return
32      sort(items, 0, len(items) - 1)
33
34  l = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
35  quicksort(l)
36  print('Liste triée:   ', l)
```

4 Chaînes de Markov

Question 5: (*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.

1. 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

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

   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.

2. 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 :

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

   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 est de 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, $\sum_i p_{ij} = 1$ (i.e chaque élément de la matrice est supérieur ou égal à 0, et la somme des probabilités à chaque ligne est toujours égale à 1).

3. 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 appelée 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 appelée *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 7 et 8.

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

1. $P(X_0 = 0, X_1 = 1, X_2 = 2)$
   $= P(X_2 = 2|X_1 = 1, X_0 = 0)P(X_1 = 1|X_0 = 0)P(X_0 = 0)$ (Bayes, règle de chaîne)
   $= P(X_2 = 2|X_1 = 1)P(X_1 = 1|X_0 = 0)P(X_0 = 0)$ (définition de la chaîne de Markov)
   $= p_{12} \times p_{01} \times P(X_0 = 0) = 0.0 \times 0.2 \times 0.3 = 0.0$

2. $P(X_0 = 0, X_1 = 1, X_2 = 1) = P(X_0 = 0) \times p_{01} \times p_{11}$
   $= 0.3 \times 0.2 \times 0.1 = 0.006$

Question 6: (*10 minutes*) **Probabilités marginales : Python**
En utilisant la loi initiale μ_0 et la matrice **P** de la question précédente, 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.

```
1  def produit_scalaire(vec, mat):
2      # vec: liste de n elements
3      # mat: liste de n sous-listes
4      result = []
5      for i in range(len(mat[0])): #iterer sur les colonnes de la matrice
6          total = 0
7          for j in range(len(vec)): # iterer sur les elements du vecteur et les lignes de la matrice
8              total += vec[j] * mat[j][i]
9          result.append(total)
10     return result
11
12 def puissance_mat(mat, n):
13     # P: liste de listes
14     # n: integer
15     new_mat = mat
16     for i in range(n-1):
17         dot_prod = [] # produit scalaire entre chaque ligne et la matrice complète
18         ... # Partie à compléter
19     return new_mat
20
21 def prob_marginales(mu_0, P, n):
22     return ... # Partie à compléter
23
24
25 mu_0 = [0.3, 0.4, 0.3]
26 P = [[0.1, 0.2, 0.7],
27     [0.9, 0.1, 0.],
28     [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 $\mathbf{P}$! Chaque ligne de $\mathbf{P}^2$ sera donc le produit scalaire entre une ligne de $\mathbf{P}$ et $\mathbf{P}$.

Solution

$$\mu_1 = \mu_0 \mathbf{P}^1 = \begin{pmatrix} 0.42 & 0.34 & 0.24 \end{pmatrix} \tag{3}$$

$$\mu_2 = \mu_0 \mathbf{P}^2 = \begin{pmatrix} 0.37 & 0.31 & 0.32 \end{pmatrix} \tag{4}$$

```
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 7: (*20 minutes*) **Simuler une chaîne de Markov : Python**
Pour cet exercice, vous n'avez pas à utiliser les calculs des exercices précédents.

Comme 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 (le nombre d'iterations est arbitraire).

Écrivez une fonction simple afin d'implémenter l'algorithme ci-dessus. Nommez-la **sim_markov()**, celle-ci prend comme paramètres **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

    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

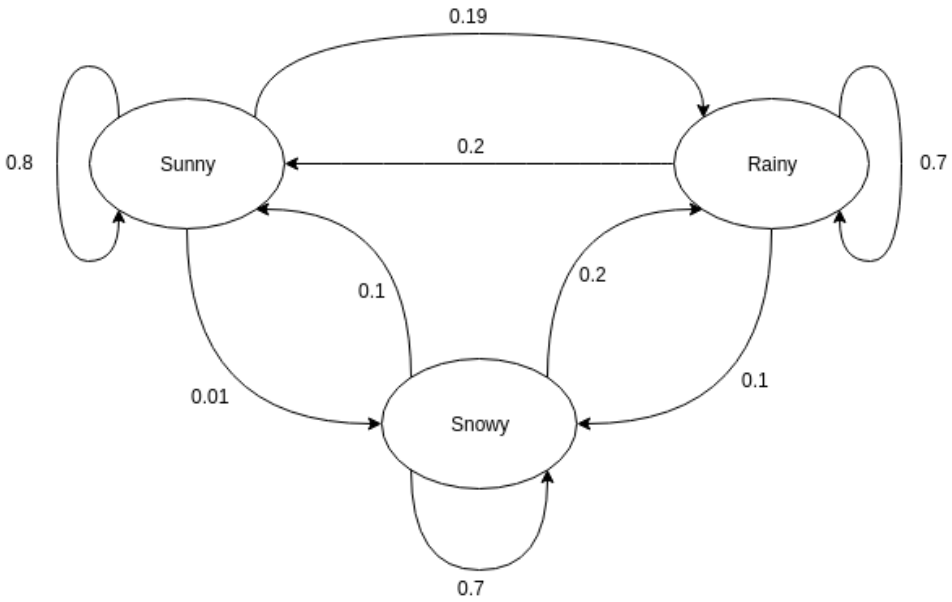
Pensez à utiliser la méthode **random.choices()**. Elle vous permet de sélectionner de façon (pseudo-)aléatoire des éléments d'une liste. N'hésitez pas à vous référer à la documentation officielle pour plus de détails.

Solution

```
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]
    print(random.choices(states, mu_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_markov(mu_0, P))
```

Question 8: (*20 minutes*) **Coder une chaîne de Markov avec un dictionnaire Python**
Cette fois-ci, vous allez utiliser un dictionnaire au lieu de vecteurs et matrices !
Supposez qu'il y'ait trois choix d'états avec les transitions dans l'image ci-dessous. Supposez également que la loi initiale des états est (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

```
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.

```
import random

def sim_markov_dict(mu_0, P, n_iters):
    # liste d'etats
    states = list(mu_0.keys())
    # premier etat
    current_state = ... # Partie à compléter
    future_states = []
    ... # Partie à compléter

    return ... # Partie à compléter

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

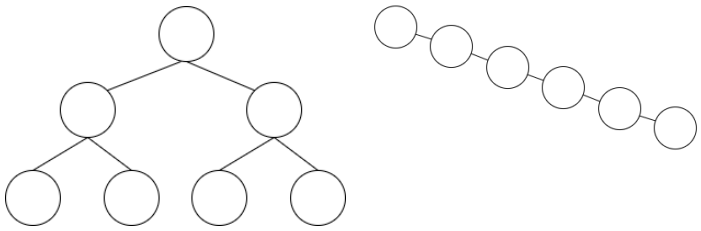
De nouveau, pensez à utiliser la méthode **random.choices**().

Solution

```
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)
```

5 Treap

Question 9: (🕒 *20 minutes*) **Insertion dans un Treap : Python**
Un arbre binaire de recherche montre de meilleures performances lorsque l'arbre est équilibré. Ainsi, la complexité d'une opération de recherche, d'insertion ou de suppression d'un nœud dans l'arbre a est de $O(logn)$ tandis qu'il sera de $O(n)$ dans l'arbre b.



(a) Arbre équilibré (b) Arbre non équilibré

FIGURE 2 – Exemple d'arbres binaires de recherche

Afin de s'assurer d'obtenir un arbre binaire de recherche équilibré, on peut utiliser les propriétés d'un heap (ou tas) dont les éléments sont ordonnés en suivant une priorité. Dans un Max-heap (Figure 3), le nœud ayant la priorité maximale se trouve toujours au sommet de l'arbre. Les nœuds parents auront toujours une priorité plus grande que les nœuds enfants. Dans un Min-heap tel que présenté en cours, le nœud ayant la plus faible priorité se trouve au sommet et dans cette configuration, les nœuds parents auront toujours un priorité plus petite que les nœuds enfants.

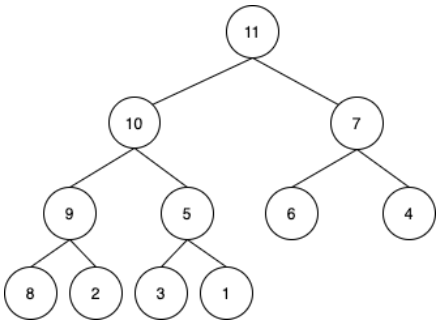


FIGURE 3 – Exemple de max-heap

Un Treap (ou arbretas) est une fusion entre un arbre binaire de recherche et un Heap. En construisant un Treap, vous devez vous assurer que les deux conditions ci-dessous soient respectées :

— Le nœud de gauche a une valeur plus petite que le nœud parent, tandis que le nœud de droite a toujours une valeur plus grande que le nœud parent. *–propriété d'un arbre binaire de recherche*
— Chaque enfant a une priorité plus petite que la priorité du parent. *–propriété d'un max-heap*

```
/* 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 */
```

Soit la liste suivante :
liste = [5, 2, 1, 4, 9, 8, 10]

À partir du fichier **treap.py** se trouvant sur Moodle, créer une nouvelle liste **nodes** qui contient des ensembles de tuples à deux éléments. Chaque tuple contiendra un élément de **liste** et une valeur aléatoire représentant une priorité.
Notez ci-dessous les valeurs que vous obtiendrez :

nodes = [(5, ...), (2, ...), (1, ...), (4, ...), (9, ...), (8, ...), (10, ...)]

Placez les éléments de **nodes** dans un Treap. Utilisez l'espace ci-dessous pour dessiner le Treap qui sera obtenu.

Figure

Conseil

La fonction **insertNode** est récursive. Inspirez-vous de l'insertion dans un arbre de recherche binaire, mais n'oubliez pas 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 nœuds enfants. **rotateLeft** et **rotateRight** permettent de réarranger les nœuds de façon à ce que la propriété de la heap soit satisfaite. Vous pouvez vous référer à l'illustration ci-dessous pour avoir une idée de comment fonctionnent les rotations.