TD 2

1 Dictionnaire, hachage, chaînage

Exercice 1.1 Programmer une fonction prenant comme paramètre un dictionnaire dont les valeurs sont des listes d'entier et renvoyant un dictionnaire dont les clés sont les différents entiers constituant les valeurs et les valeur la liste des clé possible.

Exercice 1.2 Une bibliothèque dispose d'une liste L composée des uplets ('nom', 'Livre_emprunté', date de l'emprunt). En utilisant la commande `__hash__()`, programmer une fonction prenant comme paramètre L et renvoyant le dictionnaire de clé l'entier obtenu après hachage du 'nom', et comme valeur, la liste des couples ('Livre_emprunté', date de l'emprunt), en prenant soin d'éviter toute collision par la méthode du chaînage.

2 Un peu de Récursif

Exercice 2.1 Programmer en récurssif une fonction prenant comme paramètre n et renvoyant la liste des permutations de $[[1, n]]$ sous forme de liste. Elle renverra une liste de liste $[i_1, \dots, i_n]$ où les i_k sont des entiers distincts de $[[1, n]]$ représentant la permutation σ vérifiant $\sigma(k) = i_k$.

Exercice 2.2 Pour tout entier $n \geq 2$, on note I_n la matrice identité d'ordre n , J la matrice de $\mathcal{M}_n(\mathbb{R})$ exclusivement composée de 1 et X_0 la matrice colonne $\begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} \in \mathcal{M}_{n,1}(\mathbb{R})$.

On désigne par :

- $\mathcal{U}_n$ la famille (finie) des matrices de $\mathcal{M}_n(\mathbb{R})$ constituées exclusivement de 0 et de 1 et contenant exactement deux valeurs 1 dans chaque ligne et dans chaque colonne. On note $u_n = \text{Card}(\mathcal{U}_n)$.
- $\mathcal{H}_n(i, j)$ le sous-ensemble de $\mathcal{U}_n$ formé des matrices dont le coefficient de la ligne i et colonne j vaut 1.

Pour $(i, j) \in [[1, n]]^2$, on note $S_{i,j}$ la matrice obtenue en permutant la i -ème colonne et la j -ème colonne de I_n .

1. Programmer une fonction prenant comme paramètre les entiers i, j et une matrice M de $\mathcal{M}_n(\mathbb{R})$ et renvoyant $MS_{i,j}$ et $S_{i,j}M$. Calculer $S_{i,j}^2$. Soit $M \in \mathcal{M}_n(\mathbb{R})$; décrire les matrices $MS_{i,j}$ et $S_{i,j}M$.
2. Programmer une fonction prenant comme paramètre une matrice M renvoyant true si elle est dans $\mathcal{U}_n$, False sinon.
3. Programmer une fonction prenant comme paramètre un entier n et renvoyant une liste de couple $[(i, j), (k, l)]$, $(i, j, k, l) \in [[1, n]]^4$, $i < j$ et $k < l$.
4. Proposer un programme Python prenant pour paramètre un entier n et renvoyant la liste des éléments de $\mathcal{U}_n$
5. Programmer une fonction python prenant comme paramètre un entier n renvoyant aléatoirement une matrice de $\mathcal{U}_n$.
6. Programmer une fonction python prenant comme paramètres trois entiers n, i, j renvoyant aléatoirement une matrice de $\mathcal{H}_n(i, j)$.

3 Graphe

Exercice 3.1 Soit G le dictionnaire de la liste d'adjacence d'un graphe G , non orienté, non pondéré de noeuds étiqueté par des entiers.

1. Programmer la fonction $voisin(G, x, y)$ qui renvoie $True$ si x et y sont voisins et $False$ sinon.
2. Programmer la fonction $Ajoute(G, x, y)$ qui modifie le dictionnaire G en ajoutant l'arrête (x, y) si elle n'existait pas déjà.
3. Programmer une fonction renvoyant la liste des arrêtes de G sans répétition.

Exercice 3.2 Soit G le dictionnaire de la liste d'adjacence d'un graphe G , non orienté, non pondéré de noeuds étiqueté par des entiers.

1. Programmer une fonction $Couleur(G, k, s, t)$ qui renvoie, pour un entier k une liste c de taille n (le nombre noeud de G) d'entier aléatoire de 1 à k suivant une loi uniforme (endehors de s et t) avec comme condition $c[s] = 0$, $c[t] = k + 1$. Ainsi $c[i]$ sera la couleur du noeud i .
2. Programmer une fonction $Voicincolor(G, c, i, j)$ renvoyant la liste des noeud voisin de i de couleur j où c est la liste des couleurs des noeuds.
3. Programmer $Voisinliste(G, L, c, i)$ prenant comme paramètre une liste de noeud du graphe et renvoyant la liste (sans redondance) des noeuds de couleur i voisin d'un des noeuds de L .
4. Programmer $Chemin(G, c, k, s, t)$ qui, renvoie $True$ s'il existe une chemin $s - v_1 - \dots - v_k - t$ tel que $c[v_j] = j$ et $False$ sinon.

4 Programmation dynamique

Exercice 4.1 La distance de Levenshtein est une distance, au sens mathématique du terme, donnant une mesure de la différence entre deux chaînes de caractères. Elle est égale au nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer pour passer d'une chaîne à l'autre.

On appelle distance de Levenshtein entre deux chaînes M et P le coût minimal pour transformer M en P en effectuant les seules opérations élémentaires (au niveau d'un caractère) suivantes (on agit exclusivement sur M et successivement sur ses transformés) :

- substitution ;
- insertion (ou ajout) ;
- suppression (ou effacement).

Formellement, on définit cette distance, avec deux chaînes, $\mathbf{a}$ et $\mathbf{b}$, où on note $l(\mathbf{a})$ et $l(\mathbf{b})$ la longueur de ces chaînes de caractères, $\mathbf{a}-1$ et $\mathbf{b}-1$ les chaîne privé du premier caractère.

$$lev(a, b) = \begin{cases} \max(l(a), l(b)) & \text{si } \min(l(a), l(b)) = 0 \\ lev(a - 1, b - 1) & \text{si } a[0] = b[0] \\ 1 + \min(lev(a - 1, b), lev(a, b - 1), lev(a - 1, b - 1)) & \text{sinon} \end{cases}$$

On associe à chacune de ces opérations un coût. Généralement, le coût est égal à 1 pour les trois opérations. La distance est la somme des coûts des opérations effectuées. On peut montrer que définir une distance au sens mathématique du terme entre les caractères d'un alphabet, entraîne que la distance de Levenshtein soit aussi une distance au sens mathématique du terme, sur l'ensemble des chaînes construites sur l'alphabet.

Proposer une programmation dynamique, prenant comme paramètre $\mathbf{a}$ et $\mathbf{b}$, construisant une matrice D de taille $(l(\mathbf{a}) + 1) \times (l(\mathbf{b}) + 1)$ (mémoïsation) telle que $D[i, j]$ soit la distance entre les i derniers termes de $\mathbf{a}$ et les j derniers de $\mathbf{b}$ (sous problème).

On initialisera D par $D[i, 0] = i$ et $D[0, j] = j$.

Exercice 4.2 En informatique, l'algorithme de Floyd-Warshall est un algorithme pour déterminer les distances des plus courts chemins entre toutes les paires de sommets dans un graphe orienté et pondéré, en temps cubique au nombre de sommets.

L'algorithme de Floyd-Warshall prend en entrée un graphe orienté et valué, décrit par une matrice d'adjacence donnant le poids d'un arc lorsqu'il existe et la valeur $+\infty$ sinon. Le poids d'un chemin entre deux sommets est la somme des poids sur les arcs constituant ce chemin. Les arcs du graphe peuvent avoir des poids

négatifs, mais le graphe ne doit pas posséder de cycle de poids strictement négatif. L'algorithme calcule, pour chaque paire de sommets, le poids minimal parmi tous les chemins entre ces deux sommets.

On suppose que les sommets de G sont $\{1, 2, 3, 4, \dots, n\}$. Il résout successivement les sous-problèmes suivants :

$W_{i,j}^k$ est le poids minimal d'un chemin du sommet i au sommet j n'empruntant que des sommets intermédiaires dans $\{1, 2, 3, \dots, k\}$ s'il en existe un, et $+\infty$ sinon. On note W^k le tableau des $W_{i,j}^k$. Pour $k = 0$, W^0 est la matrice d'adjacence définissant G . Maintenant, pour trouver une relation de récurrence, on considère un chemin p entre i et j de poids minimal dont les sommets intermédiaires sont dans $\{1, 2, 3, \dots, k\}$. De deux choses l'une :

- soit p n'emprunte pas le sommet k ;*
- soit p emprunte exactement une fois le sommet k (car les circuits sont de poids positifs ou nuls) et p est donc la concaténation de deux chemins, entre i et k et k et j respectivement, dont les sommets intermédiaires sont dans $\{1, 2, 3, \dots, k-1\}$.*

L'observation ci-dessus donne la relation de récurrence :

$$W_{i,j}^k = \min_{(i,j) \in [1,n]^2} (W_{i,j}^{k-1}, W_{i,k}^{k-1} + W_{k,j}^{k-1})$$

pour tous i, j et k dans $\{1, 2, 3, 4, \dots, n\}$. Ainsi on résout les sous-problèmes par valeur de k croissante.

On supposera que le graphe G , nous est donné sous la forme d'une liste d'adjacence stocker dans un dictionnaire G , où $G[i]$ renvoie la liste des uplets (v_j, p_j) de telle sorte que p_j soit le poids (relatif de l'arrête de u_i vers v_j).

- 1. Programmer une fonction `Adj(G)` retournant sous forme de tableau, la matrice d'adjacence évoquée ci-dessus et prenant comme paramètre le dictionnaire G de la liste d'adjacence du graphe (on prendra garde au fait que la ligne $i = 0$ est la première ligne).*
- 2. Proposez une fonction `cycle(A)`, renvoyant `true` si la matrice d'adjacence n'a aucun cocycle de longueur 1 de poids négatif et `False` sinon.*
- 3. Programmer une fonction `FW(A, k)` prenant comme paramètre le tableau A de la matrice d'adjacence et renvoyant W^k et s'arrête si le graphe procède un cocycle de poids négatif.*
- 4. Terminer le programme de la fonction `FloydWarshall(G)` renvoyant la distance du plus courts chemin.*