Interrogation d'informatique n°7. *Barème sur 24 pts*

Exercice A. Implémentation d'une structure de file d'attente par deux piles [3 pts]

Dans une file d'attente, l'ordre de sortie correspond à l'ordre d'arrivée. On peut implémenter **une file d'attente** f à l'aide de **deux piles (listes)** : une des listes est utilisée pour rajouter des éléments dans la file, l'autre pour enlever des éléments. On définit ainsi une file **par un couple** (p, q), où p est la pile où on enfile des éléments et où q est la pile où on défile les éléments. Lorsqu'on veut retirer un élément de la file alors que la deuxième liste q est vide, on remplace celle-ci par la première, en la renversant (afin que les premiers entrés dans la file soient les premiers sortis).

Définir les deux fonctions suivantes :

`creer_file_vide()` : crée et renvoie la file vide

`est_vide_file(f)` : renvoie `True` ssi la file f ne contient aucun élément

Ainsi que les deux procédures suivantes :

`enfiler(x,f)`: ajoute un élément x à une file f (et ne renvoie aucune valeur)

`defiler(f)` : supprime un élément x dans f (supposée non vide) et renvoie comme valeur cet élément.

Exercice B. Copie profonde [3 pts]

On considère un dictionnaire `dico` dont **les valeurs des clés sont des listes**.

1) Supposons par exemple que `dico[0]` vaut `[1]` et `dico[1]` vaut `[2]`

On considère les instructions

`dicoBis = dico.copy() ; dicoBis[0].append(0) ; dicoBis[1] = [3]`

Expliquer brièvement pourquoi `print(dico[0],dico[1])` affiche `[1,0],[2].`

2) Pour éviter le problème de copie superficielle illustré par la valeur de `dico[0]` dans l'exemple précédent, on souhaite faire une "copie profonde".

Écrire une suite d'instructions permettant d'attribuer à `dicoBis` une copie profonde de `dico`.

Remarque : Cette instruction s'obtient en Python par la fonction `deepcopy` ici non autorisée.

Exercice C. Circuits eulériens

Soit G un graphe orienté.

On suppose que G est codé en Python par le dictionnaire des listes d'adjacence.

Un circuit est un chemin $s_0 \to s_1 \to ... \to s_p = s_0$ revenant à son origine.

On le code en Python par la liste $[s_0, s_1, ..., s_{p-1}]$.

Un circuit est dit eulérien ssi il passe une et une seule fois par chaque arête de G.

Pour $s \in S$, on note $d^+[s]$ le nombre d'arêtes issues de s et $d^-[s]$ le nombre d'arêtes se terminant en s.

Ainsi, $d^+[s]$ et le nombre d'arêtes sortant de s, et $d^-[s]$ le nombre d'arêtes entrant dans s.

Lorsque G admet un circuit eulérien, on vérifie aisément que $d^+[s] = d^-[s]$ pour tout sommet $s \in S$.

On se propose ici de montrer la réciproque : si un graphe G vérifiant la propriété $(\mathcal{P}) : \forall s \in S,\, d^+(s) = d^-(s)$ et si G est connexe, alors il existe un circuit eulérien dont on va proposer une construction.

1) [2 pts] Écrire une fonction `verif(G)` qui étant donné un graphe G supposé connexe renvoie `True` ssi $d^+[s] = d^-[s]$ pour tout sommet $s \in S$. On demande un algorithme de complexité linéaire $O(n+m)$, où n est le nombre de sommets et m le nombre d'arêtes.

2) Soit un graphe G vérifiant la propriété

$$(\mathcal{P}) : d^+(s) = d^-(s) \text{ pour tout sommet } s \in S$$

Un cycle est un circuit passant par des sommets distincts.

Considérons l'algorithme suivant pour trouver un cycle (non vide) inclus dans G :

Étape 1 : On considère un sommet $s_0 \in S$ tel que $d^+[s_0] > 0$

Étape 2 : Tant que s_k n'appartient pas à $\{s_0, ..., s_{k-1}\}$, on effectue :

- on ajoute s_k à la liste des sommets déjà parcourus
- on considère un successeur s_{k+1} de s_k et on itère la boucle avec s_{k+1}

Étape 3 : On obtient $i < k$ tel que $s_i = s_k$. On renvoie le cycle $[s_i, ..., s_{k-1}]$.

a) [2 pts] On se propose de justifier la terminaison de l'algorithme précédent. Justifier les propriétés :

(i) L'algorithme ne bloque pas lorsqu'on considère un successeur s_{k+1} de s_k.

(ii) L'algorithme termine : en un nombre fini d'étapes, la boucle de l'étape 2 termine.

b) [4 pts] Écrire une procédure-fonction `cycle(G,s0)` qui

- prend en argument un graphe G vérifiant $(\mathcal{P})$ et un sommet s_0 vérifiant $d^+(s_0) > 0$
- renvoie un cycle inclus dans le graphe G
- modifie le graphe G de sorte à supprimer dans G les arêtes du cycle.

Pour avoir la totalité des points, on proposera une fonction de complexité linéaire $O(n+m)$.

On pourra utiliser un dictionnaire de marquage.

3) *Question supplémentaire*

Soit un graphe G vérifiant la propriété $(\mathcal{P}) : d^+(s) = d^-(s)$ pour tout sommet $s \in S$.

a) On considère le graphe G' obtenu en supprimant dans G les arêtes du circuit C (mais en conservant tous les sommets). Justifier brièvement que G' vérifie la propriété $(\mathcal{P})$.

b) Soit un graphe G connexe et vérifiant la propriété $(\mathcal{P})$. Montrer que G admet un circuit eulérien.

Exercice D. Optimisation du produit matriciel

Une matrice de type (n, p) est une matrice admettant n lignes et p colonnes. **Le calcul d'un produit AB de deux matrices A et B de types respectifs (n, p) et (p, q) nécessite npq opérations élémentaires.**

1) [1 pt] Écrire une fonction `ind_min(L)` qui étant donnée une liste L d'entiers non vide renvoie le plus petit indice i tel que $L[i] = \min(L)$.

2) [1 pt] Pour effectuer un produit de trois matrices A, B et C, on peut soit effectuer $(AB)C$ soit effectuer $A(BC)$.

Montrer par un exemple que le nombre d'opérations utilisées n'est pas le même selon le choix effectué.

3) [4 pts] Soit $P = [p_0, p_1, ..., p_n]$ une liste d'entiers non nuls.

Soit $(A_1, ..., A_n)$ une suite de matrices telles que $\forall k \in [\![1, p]\!]$, A_k est de type (p_{k-1}, p_k).

On souhaite calculer le produit $A_1A_2....A_n$ en effectuant $(n-1)$ produits de deux matrices.

On cherche à effectuer le minimum d'opérations élémentaires possibles.

Écrire une fonction `minimum(P)` qui prend en argument la liste P et renvoie le nombre minimum d'opérations.

On utilisera le principe de la programmation dynamique pour avoir une complexité polynômiale.

Indication : Le calcul de $A_i...A_j$ se ramène à effectuer un produit MN, où $M = A_i...A_k$ et $N = A_{k+1}...A_j$ sont supposés connus, avec $i \leq k \leq j-1$. On utilisera une fonction auxiliaire récursive qui prend en arguments des entiers i et j, avec $i < j$, et qui renvoie le nombre minimum d'opérations nécessaires pour calculer $A_i...A_j$. Le principe de mémoïsation permet d'obtenir une complexité raisonnable.

4) [1 pt] Déterminer la complexité de `minimum(P)` exprimée en fonction de n.

5) [2 pts] Pour indiquer les opérations effectuées, on utilise une formule parenthésée.

On représente chaque parenthèse ouvrante par un `"1"` et une parenthèse fermante par un `"-1"`.

Par exemple, le produit $((A_1A_2)(A_3(A_4A_5)))$ est codé par la liste d'entiers `[1,1,-1,1,1,-1,-1,-1]`.

Et le produit $(A_1(A_2(A_3(A_4A_5))))$ est codé par la liste d'entiers `[1,1,1,1,-1,-1,-1,-1]`.

Et (A_1A_2) et A_1 sont codés par `[1,-1]` et `[]`. Il y a $(n-1)$ couples de parenthèses.

Écrire une fonction `solution(P)` qui renvoie une liste représentant une configuration optimale.

6) (★) *Question supplémentaire.* Montrer que ce problème d'optimisation se ramène à la recherche d'un plus-court chemin dans un graphe pondéré positivement qu'on explicitera sans justification.

Indication : Associer à chaque sommet une liste $[i_1, ..., i_p]$, où $0 < i_1 < i_2 < ... < i_p < n$.