DS 2

Informatique de tronc commun, classe de PC

Julien Reichert

Ce devoir consiste en deux parties indépendantes comprenant toutes les deux des questions de programmation dynamique, sachant que la deuxième partie contient beaucoup de SQL.

Partie 1 : Sommes sur un chemin dans une matrice

On rappelle la consigne universelle : une fonction qui « détermine si . . . » doit renvoyer un booléen.

Sauf mention explicite du contraire, une fonction qui demande de calculer une complexité attend la complexité en temps dans le pire des cas.

Soit une liste (non vide) de listes d'entiers positifs, toutes de même taille non nulle (propriété qu'on supposera vraie dans toutes les questions sans la vérifier explicitement), qu'on appellera dans toute cette partie une matrice. Le nombre de listes n'est pas forcément leur taille commune (les matrices ne sont pas forcément carrées).

Un **chemin** dans une matrice est une suite de couples $(i_0, j_0), (i_1, j_1), \ldots, (i_n, j_n)$ telle que :
- $i_0 = j_0 = 1$;
- i_n est le nombre de lignes et j_n est la taille commune des listes ;
- $\forall 0 \leq k < n$, soit $i_{k+1} = i_k$ et $j_{k+1} = j_k + 1$, soit $i_{k+1} = i_k + 1$ et $j_{k+1} = j_k$.

En bref, on part d'en haut à gauche et on finit en bas à droite en signalant à la combientième ligne et à la combientième colonne on se situe.

La **valeur** d'un chemin est la somme des éléments de la matrice dans les cellules traversées par ce chemin.

Dans cette partie, nous allons écrire différents algorithmes de calcul de la valeur maximale d'un chemin.

Préliminaires

Question 1 : Écrire une fonction en Python qui prend en argument une matrice et qui détermine la taille commune de tous les chemins, définie comme le nombre de cases traversées.

Question 2 : Écrire une fonction en Python qui prend en argument une liste contenant des couples et qui détermine s'il peut s'agir d'un chemin dans une matrice dont les dimensions seraient conformes.

Pour simplifier la représentation et utiliser moins d'espace, on peut remplacer la suite des indices par une liste de booléens, en décidant qu'un `True` signale qu'on va à droite et qu'un `False` signale qu'on descend. Une telle liste de booléens sera appelée désormais un **itinéraire**.

Question 3 : Écrire une fonction en Python qui prend en argument une liste contenant des couples et qui renvoie l'itinéraire correspondant selon cette convention. On admettra sans le vérifier que la liste représente bien un chemin.

Question 4 : Écrire aussi la fonction réciproque de celle de la question précédente.

Enfin, une liste de booléens peut aussi correspondre à un entier écrit en binaire, en sommant tous les 2^i pour les indices i où la liste contient un `True`.

Question 5 : Écrire une fonction en Python qui prend en argument une liste de booléens et qui renvoie l'entier représenté.

Question 6 : Écrire aussi la fonction réciproque de celle de la question précédente. Un deuxième argument sera à fournir : la taille de la liste à retourner.

Question 7 : Écrire une fonction en Python qui prend en argument une matrice et un itinéraire et qui détermine si cet itinéraire peut correspondre à un chemin acceptable dans la matrice en termes de dimensions.

Question 8 : Écrire une fonction en Python qui prend en argument une matrice et un itinéraire et qui renvoie la valeur du chemin que l'itinéraire représente. On admettra sans le vérifier que l'itinéraire correspond à un chemin acceptable.

Exploration exhaustive

Question 9 : Écrire en quelques lignes une fonction en Python qui prend en argument une matrice et qui retourne la plus grande valeur possible d'un chemin dans la matrice. L'algorithme utilisé testera tous les chemins. On procèdera préférablement avec une récursion.

Question 10 : Déterminer la complexité de la fonction précédente. Commenter.

Algorithme glouton, première version

Question 11 : Écrire une fonction en Python qui prend en argument une matrice et qui retourne la valeur d'un chemin dans la matrice obtenu par l'algorithme glouton suivant : à chaque étape, si on ne peut pas aller à droite (resp. en bas), on va en bas (resp. à droite), et entre deux choix on va à l'endroit où la valeur est la plus grande (en cas d'égalité disons qu'on descend).

Question 12 : Déterminer la complexité de la fonction précédente. Commenter.

Question 13 : Justifier par un contre-exemple que la valeur obtenue n'est pas forcément optimale.

Algorithme glouton, deuxième version

Question 14 : Écrire une fonction en Python qui prend en argument une matrice et qui retourne la valeur d'un chemin dans la matrice obtenu par l'algorithme glouton suivant : on se force à passer par la case où se situe le maximum de la matrice en-dehors des coins haut-gauche et bas-droite (en cas d'égalité la plus haute et à hauteur égale la plus à gauche), et pour le reste on utilise la méthode de l'algorithme glouton précédent.

Question 15 : Déterminer la complexité de la fonction précédente. Commenter.

Question 16 : Justifier par un contre-exemple que la valeur obtenue n'est toujours pas forcément optimale.

Question 17 : Écrire une fonction en Python qui prend en argument deux couples d'indices pouvant chacun faire partie d'un chemin et qui détermine s'il existe un chemin passant par ces deux couples d'indices. On admettra sans le vérifier que ces couples sont différents.

Par exemple, la fonction répondra `False` pour les arguments `(2, 1)` et `(1, 2)`.

Question 18 : Écrire une fonction en Python qui prend en argument une matrice et un entier noté `n` et qui retourne la valeur d'un chemin dans la matrice obtenu par l'algorithme glouton suivant : on crée une liste de cases où on se force de passer, initialement vide, et on détermine au plus `n` fois de suite (tant que c'est encore possible) la case où se situe le maximum de la matrice (en résolvant les égalités comme précédemment) parmi les cases permettant au chemin à construire d'être valide et qui ne seraient pas de toute manière traversées au vu de ce qui précède, case qu'on ajoute à la liste des cases où on se force de passer, et pour le reste on utilise encore la méthode de l'algorithme glouton.

Question 19 : Déterminer la complexité de la fonction précédente. Commenter.

Question 20 : Justifier par un contre-exemple que la valeur obtenue n'est toujours pas forcément optimale, même pour la valeur maximale de `n`.

Programmation dynamique

Nous voici arrivés à la partie la plus importante du sujet : comment avoir vite la bonne réponse.

Le principe de l'algorithme est de déterminer pour chaque case de la matrice quelle est la plus grande valeur d'un chemin qui terminerait dans cette case. Cette valeur s'obtient en additionnant l'entier stocké dans la case et la plus grande valeur d'un chemin qui terminerait dans la case du haut (si elle existe) ou de gauche (si elle existe), en prenant le maximum quand il y a deux possibilités.

Question 21 : Écrire en Python une fonction qui suit ce principe pour calculer la plus grande valeur d'un chemin.

On autorisera l'approche bottom-up (plus simple) et l'approche top-down.

Question 22 : Déterminer la complexité en temps **ainsi que la complexité en espace** de la fonction précédente, toujours dans le pire des cas. Commenter.

En pratique, on peut aussi mémoriser d'où on venait quand on calcule la plus grande valeur d'un chemin qui termine dans une case, ce qui permet de reconstituer le chemin optimal.

Question 23 : Adapter la fonction précédente pour calculer également un chemin dont la valeur est maximale, renvoyée sous la forme d'un itinéraire.

Question 24 : Peut-on améliorer la complexité en espace de l'algorithme ? Si oui, suggérer une méthode sans l'écrire.

Partie 2 : Parlons plutôt de rugby cette fois...

Plutôt que de s'appesantir sur la déception de mi-décembre, rappelons que 2022 est aussi l'année où la France a réussi un grand chelem au Tournoi des Six Nations ! Pour fêter ceci, quelques exercices sur le thème !

Quelques informations ou rappels : pour marquer des points dans un match de rugby, on peut réaliser un drop ou une pénalité (trois points dans les deux cas) ou un essai (cinq points), chaque essai pouvant être suivi d'une tentative de transformation (deux points de plus en cas de réussite). Ainsi, un score total de 23 est par exemple possible avec quatre essais non transformés et un drop ($4 \times 5 + 1 \times 3$), mais aussi deux essais transformés et trois drops ($2 \times 5 + 2 \times 2 + 3 \times 3$), trois essais dont un transformé et deux drops ($3 \times 5 + 1 \times 2 + 2 \times 3$), et d'autres combinaisons.

Exercice 1 : Écrire une fonction qui prend en argument un entier positif, vu comme un score, et qui renvoie le nombre de façons de réaliser ce score au rugby. L'ordre dans lequel les points sont marqués n'est pas important.

Considérons le dictionnaire suivant, appelé `joueurs`, qui est une variable globale : les clés sont des équipes, et les valeurs sont des listes de noms de joueurs (il n'y a aucun doublon sur l'ensemble des listes). On a par exemple dans `joueurs["France"]` les éléments `"Melvyn Jaminet"`, `"Antoine Dupont"`, `"Romain Ntamack"`, etc. En particulier, en tant que variable globale, ce dictionnaire n'a jamais besoin d'être mis en argument d'une fonction.

Exercice 2 : Écrire une fonction qui prend en argument une chaîne de caractères dont on garantit qu'elle est présente quelque part dans `joueurs` et qui renvoie l'équipe dans laquelle le joueur associé est. Déterminer la complexité de cette fonction.

En raison de la complexité en question et de l'utilité potentielle de faire de nombreux appels à la fonction, l'exercice suivant devient très pertinent...

Exercice 3 : Écrire une fonction sans argument qui construit un dictionnaire indexé par les noms de joueurs et dont les valeurs sont les équipes associées.

Considérons à présent un compte-rendu de match. La gestion chaotique fait que tous les événements ont été mélangés sans séparer les équipes. Ainsi, un glorieux essai français se situe côte à côte avec une misérable pénalité anglaise sur la ligne des vingt-deux mètres. Nous allons donc avoir des comptes-rendus sous la forme de listes dont les éléments sont le joueur ayant marqué, ce qu'il a marqué (valeurs possibles : `"essai"`, `"transformation"`, `"drop"` ou `"penalite"`) et le temps de jeu associé (on ne tiendra pas compte des prolongations), heureusement dans l'ordre chronologique. Un exemple d'élément sera alors `["Anthony Jelonch", "essai", 9]`.

Exercice 4 : Écrire une fonction prenant en argument un compte-rendu et déterminant s'il est cohérent, c'est-à-dire qu'une transformation ne peut avoir lieu que directement après un essai (le joueur peut différer, mais évidemment il doit être dans la même équipe).

Exercice 5 : Une autre vérification peut être faite en pratique, pour éviter d'avoir affaire à un compte-rendu impossible. Laquelle ? Écrire une fonction correspondante.

La vérification n'est pas que l'ordre chronologique est bien respecté. Une réponse pertinente mais différente de celle attendue sera acceptée.

Exercice 6 : Écrire une fonction prenant en argument un compte-rendu et le nom des deux équipes, dans cet ordre, et renvoyant le score final en tant que couple d'entiers. On supposera sans le vérifier que le compte-rendu est cohérent.

Après un match du Tournoi des Six Nations, les équipes marquent un certain nombre de points : quatre points pour une victoire, deux points pour un match nul, aucun point pour une défaite. À cela s'ajoute cependant le bonus défensif d'un point en cas de défaite par au plus sept points d'écart et le bonus offensif quel que soit le résultat si au moins quatre essais ont été marqués (les deux équipes peuvent l'obtenir, dans ce cas).

Exercice 7 : Écrire une fonction prenant en argument un compte-rendu et le nom des deux équipes, dans cet ordre, et renvoyant le nombre de points marqués en tant que couple d'entiers. On supposera sans le vérifier que le compte-rendu est cohérent.

La suite de l'épreuve se fera en SQL, avec la base de données suivante :

- Table `Equipe`, avec pour attributs `equipe_id` (entier) et `equipe_nom` (chaîne de caractères).
- Table `Joueur`, avec pour attributs `joueur_id` (entier), `joueur_nom` (chaîne de caractères) et `equipe_id` (entier).
- Table `Match`, avec pour attributs `match_id` (entier), `match_date` (chaîne de caractères), `match_equipe1` (entier) et `match_equipe2` (entier).
- Table `Deroulement`, avec pour attributs `match_id` (entier), `joueur_id` (entier), `match_evenement` (chaîne de caractères) et `match_moment` (entier).
- Table `Points`, avec pour attributs `match_evenement` (chaîne de caractères) et `points_points` (entier).

Explications :

- Les attributs nommés `id` préfixés par le nom de la table où ils figurent sont des clés primaires (dans les autres cas ce sont des clés étrangères y faisant référence).
- La plupart des attributs correspond aux informations utilisées précédemment en Python, notamment `match_evenement`.
- Les attributs `match_equipe1` (entier) et `match_equipe2` sont aussi des clés étrangères et correspondent à `equipe_id`.
- La date d'un match est donnée sous la forme `AAAA-MM-JJ`, mais cela ne nous intéresse pas pour ce sujet.
- La table `Points` sert à faciliter le travail des dernières requêtes tout en économisant de la mémoire en ne mettant pas l'information dans chaque enregistrement de la table `Deroulement`.

Exercice 8 : Écrire une requête qui affiche tous les noms des équipes sans autre attribut.

Exercice 9 : Écrire une requête qui affiche le nom de tous les joueurs de l'équipe de France sans autre attribut.

Exercice 10 : Écrire une requête qui affiche le nom de tous les joueurs de l'équipe de France ayant marqué au moins un essai.

Exercice 11 : Écrire une requête qui affiche le moment le plus tôt et le moment le plus tardif où des points ont été marqués sur l'ensemble des matchs.

Exercice 12 : Écrire une requête qui affiche le résultat du match France - Angleterre sous la forme d'une table virtuelle contenant deux enregistrements et ayant deux attributs : `equipe_nom` et l'attribut virtuel qu'on nommera `score`. On admettra que les deux équipes ont marqué pour ne pas avoir à compliquer la requête.