DS 2

Informatique de tronc commun, classe de PC

Julien Reichert

Ce devoir consiste en des questions de cours, une partie de programmation (les dictionnaires peuvent être recommandés ou non, en fonction de l'exercice) et un problème avec de la programmation dynamique et des questions de SQL éparpillées (ces dernières peuvent être faites indépendamment ou ignorées sans préjudice du reste du problème).

Partie 1 : Questions de cours

Question 1.1 : Définir les notions de clé, clé primaire et clé étrangère. On pourra illustrer ces notions à l'aide de la base de données de la partie 3.

Question 1.2 : Donner la différence entre `WHERE` et `HAVING`.

Question 1.3 : Lorsque `d` est un dictionnaire, quelles sont les valeurs successives de `x` dans une boucle `for x in d` ?

Question 1.4 : Expliquer le principe de la programmation dynamique.

Partie 2 : Programmation

Exercice 2.1 : Écrire une fonction prenant en argument un entier strictement positif et retournant le plus grand entier strictement positif utilisant les mêmes chiffres.

Exercice 2.2 : Écrire une fonction prenant en argument une liste de couples formés par une chaîne de caractères et un nombre et retournant une liste de couples ayant pour premiers éléments chaque chaîne (n'apparaissant plus qu'une fois) et pour seconds éléments la différence entre le maximum et le minimum des nombres ayant été associés à la chaîne en question dans la liste de départ. L'ordre des couples dans la liste retournée n'importe pas.

Par exemple, pour la liste `[("a", 4), ("c", 2), ("a", 0), ("b", 1), ("c", 9), ("a", 3)]`, la fonction retournera `[("a", 4), ("c", 7), ("b", 0)]` (à l'ordre près).

Exercice 2.3 : Écrire une fonction prenant en argument une liste `lili` de listes de nombres et retournant une liste `reprep` formée par des listes d'indices où les listes figurant dans `lili` sont identiques. On ne se servira pas du test d'égalité entre des listes (on pourra écrire une fonction de même effet, mais en utilisant une boucle). Une liste dans `lili` n'apparaissant qu'une fois ne sera pas mentionnée. L'ordre dans `reprep` ainsi que dans ses sous-listes n'importe pas. Par ailleurs, il est permis de supposer que la taille des listes dans `lili` est toujours la même (mais pas de supposer que la valeur constante est par exemple trois).

Par exemple, pour la liste `[[2, 1, 2], [2, 1, 1], [1, 1, 1], [2, 1, 2], [2, 1, 1], [2, 1, 2]]`, la fonction retournera `[[0, 3, 5], [1, 4]]`.

Exercice 2.4 : Écrire une fonction prenant en argument deux listes qui sont des permutations l'une de l'autre, à ceci près que la première liste a un élément en plus, et qui retourne l'élément qui figure en plus.

Exercice 2.5 : Même exercice en supposant les listes strictement croissantes et en obligeant à avoir une complexité constante en espace (donc les dictionnaires sont interdits). La complexité devra être aussi faible que possible.

Exercice 2.6 : Écrire une fonction prenant en argument un dictionnaire et une valeur et retournant la liste (éventuellement vide) des clés associées à cette valeur. L'ordre n'importe pas.

Exercice 2.7 : Réaliser une structure de table de hachage dynamique, en tant que dictionnaire (à défaut de faire de la programmation orientée objet) ayant pour clés `n` (la taille effective dans le dictionnaire), `tab` (le tableau support, dont la taille est accessible directement mais constitue normalement l'une des valeurs caractéristiques d'une table de hachage), `f` (la fonction de hachage qui peut renvoyer un entier arbitraire, la division euclidienne se faisant après). Dans cette structure, le tableau support voit sa capacité doubler dès qu'elle n'est plus supérieure ou égale à dix fois la taille de la table de hachage. La fonction de hachage, quant à elle, ne change pas.

Partie 3 : Problème

Le thème de ce problème est le Père Noël secret.

On considèrera dans tout le problème une liste `participants` contenant des chaînes de caractères deux à deux distinctes.

Organiser un Père Noël secret revient à attribuer à chaque participant(e) un nom déterminant à qui il ou elle doit faire un cadeau.

Une façon de procéder est de faire piocher au hasard parmi les noms restants à chaque fois.

Le résultat sera dans tous les cas un dictionnaire indexé par les éléments de `participants` et dont les valeurs seront aussi issues de `participants`.

Exercice 3.1 : Écrire une fonction réalisant ce tirage au sort.

Exercice 3.2 : Écrire une fonction déterminant si pour le tirage au sort obtenu quelqu'un est amené à s'offrir un cadeau à soi-même (c'est-à-dire un point fixe, en voyant le tirage au sort comme une permutation).

Exercice 3.3 : Écrire une fonction déterminant si le tirage au sort contient au moins un couple de personnes devant se faire mutuellement un cadeau (c'est-à-dire une transposition, en voyant le tirage au sort comme une permutation).

Exercice 3.4 : Écrire une fonction déterminant si le tirage au sort consiste en un cycle, en le voyant comme une permutation.

Pour en revenir à la vérification que personne n'est amené à se faire un cadeau à lui-même, une méthode permettant d'exclure ceci (à partir d'au moins deux participants) revient à construire un cycle. Pour ceci, il suffit de créer un mélange de la liste `participants` et de considérer que chaque personne offrira un cadeau à la personne juste après dans le mélange obtenu (et le dernier au premier).

Exercice 3.5 : Écrire une fonction réalisant cette opération.

Pour information, le nombre de permutations qui sont des cycles est $(n-1)!$, ce qui est inférieur au nombre de permutations sans point fixe, qui est de l'ordre de $\frac{n!}{e}$.

Exercice 3.6 : Écrire une fonction mutant un tirage au sort pour permettre à quelqu'un, dont le nom est stocké dans un argument nommé `p`, de se retirer du Père Noël secret. Ce faisant, la personne qui devait offrir un cadeau à `p` l'offrira à la personne à qui `p` devait offrir un cadeau.

Si ce faisant, une personne est amenée à s'offrir un cadeau à elle-même, on maintient malgré tout la modification. En pratique, si on part d'un cycle, la modification ne pose aucun souci (sauf s'il ne restait que deux personnes, ce qui serait très triste...).

On observe que la complexité dans le pire des cas de la fonction précédente n'est pas constante, même en admettant que récupérer un élément d'un dictionnaire l'est.

Pour optimiser ceci, on va considérer un dictionnaire double, en tant que couple de dictionnaires : l'un indexé normalement par les clés prévues, et l'autre ayant pour clés les valeurs du premier et de sorte que si une clé est associée à une valeur dans le premier, c'est le contraire dans le deuxième.

Puisque l'on ne considère ici que des permutations, cela ne pose aucun problème en pratique.

Exercice 3.7 : Écrire une fonction de création du double dictionnaire à partir d'un dictionnaire donné.

Passons désormais à quelques questions de SQL.

La base de données est :

— Table `PARTICIPANTS`, avec les attributs `id_participant`, `identite` et `genre`.
— Table `OBJETS`, avec les attributs `id_objet`, `categorie` et `prix`.
— Table `CADEAUX`, avec les attributs `participant`, `destinataire`, `cadeau` et `satisfaction`.

L'attribut `categorie` de la table `OBJETS` permet de rassembler des objets similaires mais de nom différent. L'attribut `satisfaction` est un booléen précisant si le destinataire était content de son cadeau. Le reste est transparent.

Exercice 3.8 : Écrire une requête permettant d'obtenir le prix le plus élevé d'un objet disponible.

Exercice 3.9 : Écrire une requête permettant d'obtenir le prix moyen des cadeaux effectivement offerts (en comptant un objet offert plusieurs fois à concurrence du nombre de fois qu'il aurait été offert).

Exercice 3.10 : Écrire une requête permettant d'obtenir le prix minimal d'un objet ayant plu à au moins une personne.

Exercice 3.11 : Écrire une requête permettant d'obtenir la catégorie d'objets pour laquelle le prix total des cadeaux est le plus élevé (même remarque que pour l'exercice 3.9, et une seule suffit en cas d'égalité).

Exercice 3.12 : Écrire une requête permettant d'obtenir le nombre de fois qu'une fille a offert un cadeau à un garçon (les valeurs possibles de `genre` sont `"M"` et `"F"`).

Exercice 3.13 : Écrire une requête permettant d'obtenir le nombre de fois qu'une personne a offert un cadeau de la même catégorie que celui qu'elle a reçu.

Exercice 3.14 : Écrire une requête permettant d'obtenir la plus grande différence de prix entre le cadeau offert et le cadeau reçu sous la forme de deux attributs (plus grande différence entre reçu et offert, et plus grande différence entre offert et reçu).

Revenons finalement à de la programmation (dynamique).

Pour les deux questions à venir, il est crucial que la complexité ne soit pas exponentielle en le nombre d'éléments de la liste en argument.

Pour commencer, une personne veut faire un cadeau (en pratique un ensemble de cadeaux) avec un budget limité (il s'agira d'un argument de la fonction à écrire, noté `s` et correspondant à un entier naturel). Cette personne dispose d'une liste (notée `l` et également en argument) des cadeaux possibles en tant que couples (prix, valeur), dont on observera que le nom de l'objet n'est pas mentionné pour simplifier le problème à venir. Le prix fait progresser vers le dépassement du budget, tandis que la valeur permet d'augmenter la satisfaction du ou de la destinataire (voir la base de données).

Le but de la personne voulant faire un cadeau est de maximiser la satisfaction sans dépasser le budget.

Exercice 3.15 : Écrire une fonction résolvant ce problème.

À présent, la personne qui veut offrir des cadeaux ne sait pas la valeur de chaque objet acheté, et va donc prendre un peu de tout. Ceci étant, plutôt que de prendre un maximum de choses, son but est d'atteindre exactement son budget, en ne prenant qu'au plus un exemplaire de chaque cadeau possible.

Les arguments de la fonction à écrire sont à présent le budget et la liste des prix. On continuera à ne pas utiliser de noms pour les objets, toujours dans l'optique de simplifier le travail.

Exercice 3.16 : Écrire une fonction permettant de déterminer si le budget exact peut être atteint.

Ce sera tout pour cet énoncé, bien qu'il eût été possible d'ajouter le problème du voyageur de commerce, dont la résolution classique par programmation dynamique reste en temps et en espace exponentiels (au lieu d'un temps factoriel par l'algorithme naïf, néanmoins).