DS 3

Informatique de tronc commun, classe de PC

Julien Reichert

Partie 1 : Programmation

Contexte de l'exercice 1.1 : on considère un dé à six faces usuel. Pour information ou rappel, chaque face est voisine de toutes les autres sauf celle pour laquelle la somme des valeurs est sept (par exemple 2 est voisine de 1, 3, 4 et 6 mais pas 5).

Une personne souhaite désigner successivement des faces voisines deux à deux sous trois contraintes : chaque face doit être visitée au moins une fois, d'une voisine à l'autre la parité doit changer (en particulier après la face 2 il faut désigner la face 1 ou la face 3), et la somme des faces visitées doit être 25.

Exercice 1.1 : Écrire une fonction prenant en argument la face de départ et renvoyant une liste de faces successives qui répond à la question.

Contexte des exercices 1.2 et 1.3 : afin de diversifier mes sorties repas, je dispose de la liste de mes passages au restaurant (*ce n'est pas vrai, c'est pour l'intérêt pédagogique*). Puisque je vais toujours à intervalle identique au restaurant (*idem*), on peut considérer qu'une unité de temps correspond à l'écart entre deux indices successifs de la liste. Dans cette liste, il y aura bien entendu des répétitions.

Exercice 1.2 : Écrire une fonction qui prend en argument la liste en question et qui renvoie le nom du restaurant auquel ma dernière visite remonte au plus longtemps. **Donner la complexité de cette fonction.**

Exercice 1.3 : Même exercice mais il s'agit de renvoyer une liste de couples (ou listes de taille deux) associant chaque restaurant au nombre d'unités de temps depuis mon dernier passage (avec la convention que la valeur est zéro pour le dernier restaurant où j'ai été). **Donner la complexité de cette fonction.**

Exercice 1.4 : Écrire une fonction prenant en argument une liste de nombres et un élément de la liste (pas besoin de tester l'appartenance, c'est garanti) et renvoyant le classement de l'élément en excluant toutes les occurrences du maximum de la liste (il est aussi garanti sans avoir besoin de le vérifier que l'élément en question n'est pas égal au maximum de la liste).

Attention, dans l'exercice 1.4, comme on utilise le terme de « classement », si le deuxième argument s'avère être le plus grand élément à l'exclusion des occurrences du maximum, il faudra renvoyer `1` et non `0`. De même, pour la liste `[7, 2, 5, 5, 5, 7, 7, 3]` et la valeur `3`, on renverra `4` car il y a trois valeurs supérieures à `3` quand on exclut les `7` et ces valeurs sont toutes `5`.

Exercice 1.5 : On considère une liste d'entiers de taille impaire notée `n` (variable à créer si on veut s'en servir, bien entendu) ayant la propriété suivante : en-dehors d'un élément qui n'est ni le minimum ni le maximum, tous les `n-1` autres éléments de la liste peuvent être regroupés par paires de sorte que les `(n-1)/2` paires ainsi formées soient de même somme. Écrire une fonction prenant en argument une telle liste et renvoyant l'élément « intrus ».

Exercice 1.6 : On considère deux listes de nombres de même taille (inutile de le vérifier). On cherche à localiser le maximum des éléments obtenus en prenant à chaque indice la plus petite des deux valeurs à cet indice parmi les deux listes. Écrire une fonction prenant en argument les deux listes et renvoyant l'indice où ce maximum est trouvé. En cas d'égalité n'importe quel indice correspondant pourra être renvoyé.

Exercice 1.7 : Même exercice avec deux dictionnaires. Dans ce cas, on cherche une clé maximisant le minimum des valeurs entre les deux dictionnaires, à ceci près que les tailles sont quelconques et que si une clé n'est présente que dans un des deux dictionnaires elle ne peut pas être retenue dans l'optique de la maximisation. Si aucune clé n'est commune aux deux dictionnaires, il faudra en particulier renvoyer `None`.

Exemple pour l'exercice 1.6 : avec les listes `[1, 2, 4, 8]` et `[7, 5, 3, 1]`, il faudra renvoyer `2` car les deux valeurs à l'indice `2` des deux listes sont `4` et `3`, la plus petite des deux étant `3` et à aucun autre indice on trouve deux valeurs supérieures à ceci.

Exemple pour l'exercice 1.7 : avec les dictionnaires `{ 'd' : 9 , 'e' : 8 , 'f' : 10 , 'i' : 5 , 'n' : 3 }` et `{ 'd' : 9 , 'e' : 7}`, la réponse sera `'d'`.

Partie 2 : Base de données « opéra »

Pour cet exercice, nous utiliserons une adaptation (noms changés pour l'esthétique) simplifiée de la base de données que j'ai créée pendant le confinement pour recenser les spectacles musicaux auxquels j'ai assisté, ainsi que diverses informations à leur propos.

- Table `SPECTACLE`, avec les attributs `id` (clé primaire avec auto-incrémentation), `date` (chaîne de caractères [1]), `titre` (chaîne de caractères), `compositeur` (chaîne de caractères), `pays` (chaîne de caractères) et `ville` (chaîne de caractères).
  Le couple `(date, titre)` est aussi une clé, mais pas l'attribut `date` [2].
- Table `DISTRIBUTION`, avec les attributs `id` (clé étrangère vers l'attribut du même nom dans la table `SPECTACLE`), `sur_scene` (booléen, assimilable à un entier valant forcément 0 ou 1, respectivement pour faux et vrai), `role` (chaîne de caractères) et `personne` (chaîne de caractères).
  Le couple `(id, role)` est une clé.
- Table `ACCOMPAGNEMENT`, avec les attributs `id` (clé étrangère de même) et `nom` (chaîne de caractères).

Pour clarifier, l'attribut `sur_scene` est à faux pour des valeurs de `role` telles que `"Chef d'orchestre"`, `"Metteur en scène"`, etc. On en déduit que `personne` est l'attribut correspondant à l'identité des chanteurs / acteurs / directeurs...

La table `ACCOMPAGNEMENT` liste l'identité de toutes les personnes m'ayant accompagné, et ce pour chaque spectacle.

Exercice 2.1 : Proposer une clé pour la table `ACCOMPAGNEMENT`.

Exercice 2.2 : Écrire une requête permettant d'obtenir la liste des titres des spectacles que j'ai vus en Autriche.

Exercice 2.3 : Écrire une requête permettant d'obtenir le nombre de personnes ayant été dans la distribution au moins une fois sur scène et au moins une fois pas sur scène.

Exercice 2.4 : Écrire une requête permettant d'obtenir le nombre maximal de rôles qui ne sont pas sur scène qu'une personne a pu avoir, tous spectacles confondus.

Exercice 2.5 : Écrire une requête permettant d'obtenir l'effectif maximal du groupe de personnes dans lequel j'étais lors d'un spectacle.

Exercice 2.6 : Écrire une requête permettant de déterminer si oui ou non j'ai déjà été accompagné de quelqu'un qui était impliqué dans un autre spectacle (que ce soit sur scène ou non), ou éventuellement un homonyme puisqu'on ne peut pas vraiment faire la différence.

La requête est en particulier censée retourner la valeur `1` ou `0` suivant que la réponse soit respectivement oui ou non (on assimile les booléens à ces valeurs donc il n'y a rien de particulier à faire).

Si on a du mal à produire un booléen, il reste la possibilité de regarder un peu plus loin...

Exercice 2.7 : Expliquer le résultat de la requête suivante :

```
SELECT COUNT(*)
FROM SPECTACLE AS S1 JOIN SPECTACLE AS S2 ON S1.titre = S2.titre
WHERE S1.compositeur <> S2.compositeur
AND S1.id < S2.id
```

1. ... au format JJ/MM/AAAA, pas très idéal pour trier par ordre chronologique, heureusement qu'il y a l'identifiant pour cela !
2. Je peux être très motivé certains jours...

Exercice 2.8 : Expliquer le résultat de la requête suivante :

```
SELECT COUNT(*) > 1
FROM SPECTACLE JOIN DISTRIBUTION ON SPECTACLE.id = DISTRIBUTION.id
WHERE role = "Metteur en scène"
GROUP BY titre, personne
ORDER BY COUNT(*) DESC LIMIT 1
```

Exercice 2.9 : Expliquer le résultat de la requête suivante :

```
SELECT personne
FROM SPECTACLE JOIN DISTRIBUTION ON SPECTACLE.id = DISTRIBUTION.id
WHERE role = "Chef d'orchestre"
GROUP BY personne
HAVING COUNT(DISTINCT ville) =
(
  SELECT COUNT(DISTINCT ville) AS nb_villes
  FROM SPECTACLE JOIN DISTRIBUTION ON SPECTACLE.id = DISTRIBUTION.id
  WHERE role = "Chef d'orchestre"
  GROUP BY personne
  ORDER BY nb_villes DESC LIMIT 1
)
```

Exercice 2.10 : Expliquer le résultat de la requête suivante :

```
SELECT COUNT(DISTINCT S1.titre)
FROM SPECTACLE AS S1 JOIN ACCOMPAGNEMENT AS A1 ON S1.id = A1.id
JOIN SPECTACLE AS S2 ON S1.titre = S2.titre AND S1.id < S2.id
JOIN ACCOMPAGNEMENT AS A2 ON S2.id = A2.id
WHERE A1.nom = A2.nom
GROUP BY nom
ORDER BY COUNT(DISTINCT S1.titre) DESC LIMIT 1
```

Partie 3 : Alignement de séquences génétiques et ouvertures

Problème très proche de celui de la distance d'édition, l'alignement de séquences génétiques correspond plutôt au point de vue des sciences de la vie. Il s'agit de considérer deux séquences d'ADN présumées proches et ayant divergé suite à des mutations mises en œuvre dans les processus évolutifs, et de mettre en lumière les ajouts, suppressions et modifications de nucléotides.

Exercice 3.1 : Pour commencer, et de manière totalement indépendante, écrire une fonction prenant en argument une liste de nucléotides (valant donc tous `'A'`, `'C'`, `'G'` ou `'T'`) et renvoyant une liste de couples (un nucléotide, son nombre d'apparitions) pour tous les nucléotides présents.

À présent, une petite réminiscence du TP sur la programmation dynamique...

Exercice 3.2 : Écrire une fonction prenant en argument deux séquences d'ADN (chaînes de caractères) et renvoyant le nombre minimal d'opérations à effectuer pour passer de l'une à l'autre, les opérations pouvant être le remplacement d'un élément par un autre, l'insertion d'un élément ou la suppression d'un élément.

Par exemple, la fonction renverra `3` pour `"ATTAAAC"` et `"ATAAAGG"`, une possibilité étant de supprimer un `"T"`, de remplacer le `"C"` par un `"G"` et d'ajouter un `"G"`.

Exercice 3.3 : Adapter la fonction pour renvoyer deux chaînes de caractères matérialisant l'alignement, c'est-à-dire que pour chaque caractère ajouté pour passer de la première chaîne à la deuxième, un caractère trait d'union sera mis en face du nouveau caractère dans la première, pour les suppressions c'est le contraire et pour les remplacements les deux caractères seront à la même position. Dans le cas où des caractères sont maintenus d'une chaîne à l'autre, les éléments seront identiques à l'indice en question.

Pour le même exemple et les modifications données ci-avant, les deux chaînes renvoyées pourront être `"ATTAAAC-"` et `"AT-AAAGG"`.

Exercice 3.4 : Écrire une fonction qui prend en argument une liste de chaînes de caractères (tous les caractères sont des nucléotides) et qui renvoie la « matrice » (liste de listes en pratique) formée de toutes les distances calculées deux à deux (les distances au sens de la fonction de l'exercice 3.2) entre des chaînes. L'indexation dans la matrice correspondra à l'indexation dans la liste de chaînes de caractères.

Dans l'exercice précédent, on pourra ignorer les valeurs « sur et en-dessous de la diagonale » vu que la matrice sera symétrique et de diagonale nulle.

Exercice 3.5 : Petit exercice pour prendre des points assez gratuits, écrire une fonction qui prend en argument une matrice telle qu'obtenue par la fonction précédente et qui renvoie un couple d'indices `i, j` tels que `i < j` et la valeur à cet endroit de la matrice minimise la partie au-dessus de la diagonale de la matrice.

Dans la foulée de l'exercice précédent, une méthode de construction d'un arbre phylogénétique (inutile de s'attarder sur la décision) est de fusionner à chaque étape deux éléments identifiés comme les plus proches et d'adapter la matrice aux éléments fusionnés par un calcul de distances faisant intervenir des moyennes (un exemple historique d'algorithme est « UPGMA »).

Exercice 3.6 : À quelle catégorie appartient un tel algorithme comme UPGMA, qui considère à chaque étape un minimum, quitte à ce que le résultat soit incorrect en pratique ?

Attention, l'exercice 3.6 n'est pas une question où il faut programmer !

Exercice 3.7 : Revenons dans la situation de l'exercice 3.4. Cette fois, la liste contient des couples formés d'abord par la chaîne de nucléotides et ensuite par une chaîne de caractères renseignant une espèce. On dispose en plus d'une chaîne de caractères (des nucléotides) supplémentaire pour laquelle l'espèce est inconnue et d'un entier `k`. À l'aide de l'algorithme kNN appliqué à la distance d'édition au sens de l'exercice 3.2, écrire avec tous ces arguments une fonction renvoyant l'espèce présumée pour la chaîne de nucléotides supplémentaire.

En pratique, on aurait aussi pu imaginer qu'aucune information sur les espèces à part le nombre d'espèces différentes au total n'était disponible et faire une application de l'algorithme des k-moyennes...

Partie 4 : Puissance quatre en 3D

Voici un jeu découvert il y a un peu moins d'un an. Comme le puissance quatre, il oppose deux joueurs qui tentent d'aligner quatre pions de leur couleur en plaçant chacun un jeton à tour de rôle, mais au lieu de jouer sur une grille rectangulaire de sept cases par six, le plateau de jeu est composé de seize piquets, organisés en carré, et pouvant contenir quatre jetons chacun, et toutes les directions sont possibles pour un alignement gagnant, même une diagonale du cube ainsi représenté.

La grille de jeu sera une liste de listes de listes d'entiers nommée `grille`, avec les valeurs `-1` pour un joueur, `1` pour l'autre et `0` pour les cases vides.

Comme la gravité s'applique, un jeton est lâché au niveau d'une ligne et d'une colonne particulière non encore remplie et tombe le long d'un piquet pour se poser au-dessus d'éventuels jetons déjà présents.

La première composante dans l'indexation sera la ligne, la deuxième sera la colonne, et la troisième sera la hauteur dans le piquet correspondant, ce qui veut dire que si le coup initial du joueur aux jetons `1` est dans la deuxième ligne, quatrième colonne, il y aura un `1` en `grille[1][3][0]`.

Exercice 4.1 : Écrire une fonction nommée `grille_vide`, sans argument, qui crée la grille initiale.

Exercice 4.2 : Écrire une fonction prenant en argument une grille de jeu et qui renvoie la liste des couples (indice de ligne, indice de colonne) correspondant aux piquets où l'on peut encore déposer un jeton.

Exercice 4.3 : Écrire une fonction prenant en argument une grille de jeu et qui renvoie `1` s'il y a un alignement de quatre `1` où que ce soit dans la grille, `-1` si c'est le cas pour des `-1` et `0` sinon, sachant qu'il sera impossible qu'il y ait un alignement de `1` et un alignement de `-1` à la fois.

Pour la fonction précédente, une possibilité est de considérer toutes les cases de départ d'un alignement de taille quatre, et tous les vecteurs de déplacement formés de valeurs `-1`, `0` et `1` (sauf le déplacement nul), et d'extraire les alignements (à condition qu'ils soient de taille quatre) puis de tester si tous les éléments sont identiques et non nuls.

Exercice 4.4 : Écrire une fonction prenant en argument une grille de jeu et le numéro d'un joueur (`-1` ou `1`) et qui renvoie la liste de tous les couples `(i, j)` tels qu'en jouant à la ligne et la colonne données par un tel couple le joueur en question gagne immédiatement. La valeur de retour sera en particulier vide si aucun couple ne convient. On peut admettre qu'il n'y a actuellement aucun alignement déjà formé.

Exercice 4.5 : Écrire une fonction prenant en argument une grille de jeu et le numéro d'un joueur (`-1` ou `1`) et qui renvoie la liste de tous les couples `(i, j)` tels que si le joueur puis l'adversaire jouent tous deux à la même ligne et la même colonne d'un tel couple le joueur ne gagne pas puis l'adversaire gagne en un coup. La valeur de retour sera en particulier une liste vide si aucun couple ne convient. On peut admettre qu'il n'y a actuellement aucun alignement déjà formé.

Les deux questions précédentes permettent de simuler un minmax de profondeur un et deux. Sans utiliser directement l'algorithme, on va encore ajouter un cran de profondeur.

Exercice 4.6 : Écrire une fonction prenant en argument une grille de jeu et le numéro d'un joueur (`-1` ou `1`) et qui renvoie la liste des couples `(i, j)` tel qu'en jouant à la ligne et la colonne précisées le joueur en question gagnera forcément au tour suivant (ou immédiatement), en particulier il est impossible que l'adversaire gagne à son éventuel tour suivant. La valeur de retour sera en particulier une liste vide si aucun couple ne convient. On peut admettre qu'il n'y a actuellement aucun alignement déjà formé.

Exercice 4.7 (question de cours) : En profiter pour rappeler le principe de l'algorithme minmax.

Partie 5 : Un petit jeu à information imparfaite

Les jeux à information imparfaite sont certes hors programme, mais que cela n'empêche pas de faire une rapide initiation à partir d'exercices abordables. On considère la situation suivante : un plateau contient quatre pièces disposées en carré, certaines en position pile, d'autres en position face. Une personne prétend pouvoir les faire retourner afin qu'elles soient toutes en position identique, sans les regarder ni obtenir la moindre information sur leur position, et ceci en un maximum de sept mouvements.

Une contrainte supplémentaire s'applique afin d'éviter d'explorer simplement toutes les possibilités : après chaque instruction de la personne, le plateau sur lequel les pièces reposent pourra être pivoté d'autant de quarts de tours qu'une autre personne souhaitera. La seule règle est que le jeu s'arrête quand toutes les pièces sont en position identique, et on informe la personne qu'elle a réussi.

On assimilera une configuration du jeu à un tableau de quatre booléens, représentant respectivement le fait que la pièce en haut à gauche, puis en haut à droite, puis en bas à droite, puis en bas à gauche sont en position pile.

Une instruction du joueur sera « retourne les pièces en haut à gauche et en bas à droite », par exemple. Il est inutile de retourner aucune ou quatre pièces, par ailleurs.

Exercice 5.1 : Combien y a-t-il de positions possibles ?

Exercice 5.2 : Puisque le jeu est invariant par échange des rôles de pile et face, à combien de positions peut-on se limiter (en comptant la position finale) ?

Exercice 5.3 : Puisque les rotations peuvent se faire à l'aveugle et le jeu est alors invariant par rotation, à combien de positions peut-on se limiter (en comptant encore la position finale) ?

(Sans avoir compris le principe de la question précédente, il est a priori impensable de résoudre la suite…)

Exercice 5.4 : De même, quelles sont les **trois** catégories de mouvements possibles, aux rotations et échanges de pile ou face près ?

Exercice 5.5 : Pour chaque position et chaque catégorie de mouvement, quelles sont les positions possibles quand on fait le mouvement en question à partir de la position en question ?

Exercice 5.6 : En considérant qu'au début toutes les positions sont possibles (sauf la position finale), proposer une stratégie en sept coups pour le joueur et justifier pourquoi elle fonctionne.

Partie 6 : Une énigme pour finir

Une belle illustration du minmax en tant qu'énigme entendue à un mariage l'été dernier :

Sept aimants sont alignés sur une table, certains ont le pôle Nord vers le haut, d'autres l'ont vers le bas. Sans information supplémentaire, il est inutile de chercher à distinguer les pôles Nord et Sud, et le but du jeu est d'identifier un aimant qui faisait partie de la catégorie majoritaire dans le placement initial.

Les seules opérations permises sont le test de deux aimants, permettant de déduire qu'ils étaient initialement alignés de la même façon ou de façon différente.

Plutôt que de se perdre dans un exercice d'électromagnétisme, parlons plutôt de la modélisation informatique :

On considère une liste de sept booléens, et on voudrait désigner un indice auquel l'élément de la liste est présent en majorité dans la liste, mais il est interdit de parcourir la liste, seules des comparaisons entre deux éléments sont autorisées. **Il ne s'agit pas de programmation dans cette partie !**

Exercice 6.1 : Combien faut-il de comparaisons pour répondre à la question attendue, quel que soit le résultat des comparaisons ? Donner la stratégie associée.

Bien entendu, la méthode naïve avec six comparaisons fonctionne mais n'est pas optimale.

Exercice 6.2 : Prouver en particulier qu'aucune stratégie ne fonctionne avec strictement moins de comparaisons.

Annexe : rappels de Python

Manipulations sur un dictionnaire en Python :

```
d = dict() # création
d = { cle1 : valeur1, cle2 : valeur2 } # initialisation avec par exemple deux clés
c in d # test d'appartenance en temps constant (mensonge admis)
d[c] # accès en temps constant (mensonge admis)
for c in d: # parcours des clés
    corps_de_boucle()
d.keys() # séquence contenant les clés
d.values() # séquence contenant les valeurs
d.items() # séquence contenant les couples (clés, valeurs)
len(d) # nombre de clés
v = d.pop(cle) # retire la clé du dictionnaire et met dans v la valeur qui était associée
```

Gestion des chaînes de caractères en Python : comme pour les listes **mais les chaînes ne sont pas mutables**, on peut parcourir caractère par caractère avec un `for`, parcourir indice par indice avec `range` et `len` et l'indexation usuelle. Pour mettre à jour une variable contenant une chaîne, par exemple : `s = s[:4] + 'A' + s[5:]`.

Gestion des couples en Python, ou plus généralement des n-uplets : comme pour les listes (le délimiteur est une parenthèse voire rien si ce n'est pas ambigu) mais on ne peut pas muter le contenu d'un n-uplet à un indice particulier, simplement redéfinir le n-uplet. Ainsi, si `c` vaut (2, 5), remplacer le `2` par un `4` se fait par `c = (4, 5)` et non pas par `c[0] = 4`. Rappel : ajouter un n-uplet écrit explicitement à une liste nécessite **deux parenthèses** (`append` et délimitation du n-uplet).

Listes de listes ou listes de listes de listes en Python : chaque indexation donne un élément à un cran de profondeur de plus, donc on trouvera par exemple `l[i][j][k]`.

Création d'une liste de listes (« matrice ») : il faut prendre garde à ne pas écrire `[[0] * n] * m` car cela produit `m` listes de taille `n` dépendantes (toute mutation de l'une entraîne une mutation de l'autre). On écrira plutôt `[[0] * n for _ in range(m)]`, le `* n` ne posant pas de problème car `0` est un entier et que les entiers ne sont pas mutables.

Pour tous les exercices où l'on souhaite faire un tri soi-même alors qu'aucun tri n'est explicitement demandé, on pourra utiliser `sorted` (fonction qui produit une liste qui est la copie croissante de l'argument) ou `l.sort()` (mute `l` en la triant de manière croissante, avec la possibilité d'ajouter un critère de tri mais on ne va pas s'étendre dessus dans cette annexe).