Correction du DS 3

Julien Reichert

Partie 1

Exercice 1.1

Pour avoir un total de 25 en visitant toutes les faces au moins une fois, on peut répéter en plus de toutes ces faces (total 21) les valeurs 1, 2 et 1 à la fin ou 4, la décision dépendant de la parité de la face de départ. Si la première face est impaire, la dernière du trajet visitant tout d'abord chacune des six faces est forcément paire, et on doit éviter que ce soit 6. Si la première face est paire, la dernière est impaire, et on doit éviter que ce soit 3.

Cela donne donc les solutions :

- $1-4-5-6-3-2-1-2-1$
- $2-3-6-5-4-1-4$
- $3-6-5-4-1-2-1-2-1$
- $4-5-6-3-2-1-4$
- $5-6-3-2-1-4-1-2-1$
- $6-3-2-1-4-5-4$

La programmation n'est en pratique plus de la programmation mais...

```
def solution_de(depart):
  reponse = dict()
  reponse[1] = [1, 4, 5, 6, 3, 2, 1, 2, 1]
  reponse[2] = [2, 3, 6, 5, 4, 1, 4]
  reponse[3] = [3, 6, 5, 4, 1, 2, 1, 2, 1]
  reponse[4] = [4, 5, 6, 3, 2, 1, 4]
  reponse[5] = [5, 6, 3, 2, 1, 4, 1, 2, 1]
  reponse[6] = [6, 3, 2, 1, 4, 5, 4]
  return reponse[depart]
```

Exercice 1.2

```
def dernier_restaurant(restaurants):
    noms = dict()
    reponse = ""
    for i in range(len(restaurants)-1, -1, -1):
        if restaurants[i] not in noms:
            reponse = restaurants[i]
            noms[restaurant[i]] = 1
    return reponse
```

La complexité est linéaire en considérant les opérations sur le dictionnaire comme étant en temps constant.

Exercice 1.3

```
def delai_restaurants(restaurants):
    noms = dict()
    for i in range(len(restaurants)):
        noms[restaurants[i]] = i
    reponse = []
    for nom in noms:
        reponse.append((nom, len(restaurants)-1-noms[nom]))
    return reponse
```

La complexité reste linéaire grâce aux dictionnaires.

Exercice 1.4

```
def classement_sans_major(liste, valeur):
    superieurs = 0
    indmax = 0
    occmax = 1
    for i in range(1, len(liste)):
        if liste[i] > liste[indmax]:
            indmax = i
            occmax = 1
        elif liste[i] == liste[indmax]:
            occmax += 1
        if liste[i] > valeur:
            superieurs += 1
    return 1 + superieurs - occmax
```

Exercice 1.5

Principe : On trie la liste, le plus petit plus le plus grand donne la somme à atteindre, puis on progresse vers l'intérieur depuis les deux bouts et on détecte le moment où la somme ne correspond pas.

```
def intrus(l):
    lbis = sorted(l)
    somme = lbis[0] + lbis[-1]
    deb = 0
    fin = len(lbis)-1
    while deb < fin and lbis[deb] + lbis[fin] == somme:
        deb += 1
        fin -= 1
    if deb == fin or lbis[deb+1] + lbis[fin] == somme:
        return lbis[deb]
    return lbis[fin]
```

Exercice 1.6

On peut envisager d'utiliser la fonction `zip`, qui n'est certes pas au programme, mais comme il faut renvoyer un indice cela n'allège pas le code.

Pour information quand même, `zip(l1, l2)` produit la séquence contenant les couples `(l1[i], l2[i])` jusqu'à ce qu'une des deux listes soient terminées. On peut mettre autant d'arguments qu'on veut à cette fonction, et une possibilité serait de parcourir `zip(l1, l2, range(min(len(l1), len(l2))))` en s'épargnant le `min` vu l'énoncé.

```
def maximum_des_minimas(l1, l2):
    n = len(l1)
    assert len(l2) == n # Pas demandé, mais question de principe.
    rep = 0
    mini = min(l1[0], l2[0])
    for i in range(1, n):
        x = min(l1[x], l2[x])
        if x < mini:
            rep = i
            mini = x
    return rep
```

Exercice 1.7

Une version du calcul du maximum quand on ne peut pas initialiser à la première valeur, et où on protège les comparaisons avec la vérification que l'on n'a pas encore la valeur par défaut.

```
def maximum_des_minimas_dict(d1, d2):
    rep = None
    maxi = None
    for cle in d1:
        if cle in d2:
            x = min(d1[cle], d2[cle])
            if maxi is None or x > maxi:
                maxi = x
                rep = cle
    return rep
```

Partie 2

Exercice 2.1

Comme une personne peut m'accompagner à plusieurs spectacles (donc `nom` n'est pas une clé) et que plusieurs personnes peuvent m'accompagner à un spectacle (donc `id` non plus), mais qu'il s'agit tout de même de gérer les doublons de la table, la clé sera `(id, nom)`.

Exercice 2.2

```
SELECT DISTINCT titre FROM SPECTACLE
WHERE pays = 'Autriche'
```

On peut éventuellement retirer `DISTINCT` sans contredire directement la consigne...

Exercice 2.3

Version la plus belle mais compliquée :

```
SELECT COUNT(*) FROM
(
  SELECT personne FROM DISTRIBUTION
  GROUP BY personne
  HAVING MAX(sur_scene) = 1 AND MIN(sur_scene) = 0
)
```

Version plus classique :

```
SELECT COUNT(*) FROM
(
  SELECT DISTINCT personne FROM DISTRIBUTION WHERE sur_scene
  INTERSECT
  SELECT DISTINCT personne FROM DISTRIBUTION WHERE NOT sur_scene
)
```

Exercice 2.4

```
SELECT COUNT(DISTINCT role) FROM DISTRIBUTION
WHERE NOT sur_scene
GROUP BY personne
ORDER BY COUNT(DISTINCT role) DESC LIMIT 1
```

Exercice 2.5

```
SELECT 1 + COUNT(*) FROM ACCOMPAGNEMENT
GROUP BY id
ORDER BY COUNT(*) DESC LIMIT 1
```

On n'oubliera pas d'ajouter `1` pour moi !

Exercice 2.6

```
SELECT COUNT(*) > 0 FROM DISTRIBUTION JOIN ACCOMPAGNEMENT
WHERE personne = nom
```

La réponse est non, au passage.

Exercice 2.7

Cette requête donne le nombre de fois où j'ai assisté à deux spectacles de même titre mais de compositeurs différents (si cela arrive, cela peut d'ailleurs être compté en plusieurs exemplaires). La réponse est en pratique zéro aussi.

Exercice 2.8

Cette requête détermine s'il est arrivé au moins une fois que je voie au moins deux fois un spectacle d'un certain titre, avec en particulier au moins deux fois le même metteur en scène. La réponse est oui, mais je préfère ne pas en parler.

On notera que cette requête est censée aider à rédiger la requête de l'exercice 2.6.

Exercice 2.9

Cette requête donne le chef d'orchestre que j'ai vu diriger dans le plus de villes différentes, en donnant tous les noms en cas d'égalité. Malheureusement, telle quelle la requête donne `(pas d'orchestre)`, donc il faudrait exclure cette dernière valeur pour avoir l'information…

Exercice 2.10

Cette requête donne le nombre de titres d'œuvres que j'ai vues au moins deux fois avec une même personne, en ne comptant que pour la personne avec qui c'est arrivé le plus souvent.

Partie 3

Exercice 3.1

```
def nombre_apparitions(chaine):
    compte = dict()
    for car in chaine:
        if car not in compte:
            compte[car] = 1
        else:
            compte[car] += 1
    return list(compte.items())
```

Exercice 3.2

```
def distance_edition(s, t):
    ns, nt = len(s), len(t)
    distances = [[None for j in range(nt+1)] for i in range(ns+1)]
    for i in range(ns+1):
        distances[i][0] = i
    for j in range(nt+1):
        distances[0][j] = j
    for i in range(1, ns+1):
        for j in range(1, nt+1):
            distances[i][j] = distances[i-1][j] + 1
            if distances[i][j] > distances[i][j-1] + 1:
                distances[i][j] = distances[i][j-1] + 1
            if distances[i][j] > distances[i-1][j-1] + (s[i-1] != t[j-1]):
                distances[i][j] = distances[i-1][j-1] + (s[i-1] != t[j-1])
    return distances[-1][-1]
```

Exercice 3.3

```
def distance_edition_bis(s, t):
    ns, nt = len(s), len(t)
    distances = [[None for j in range(nt+1)] for i in range(ns+1)]
    directions = [[None for j in range(nt+1)] for i in range(ns+1)]
    for i in range(ns+1):
        distances[i][0] = i
        directions[i][0] = (-1, 0)
    for j in range(nt+1):
        distances[0][j] = j
        directions[0][j] = (0, -1)
    for i in range(1, ns+1):
        for j in range(1, nt+1):
            distances[i][j] = distances[i-1][j] + 1
            directions[i][j] = (-1, 0)
            if distances[i][j] > distances[i][j-1] + 1:
                distances[i][j] = distances[i][j-1] + 1
                directions[i][j] = (0, -1)
            if distances[i][j] > distances[i-1][j-1] + (s[i-1] != t[j-1]):
                distances[i][j] = distances[i-1][j-1] + (s[i-1] != t[j-1])
                directions[i][j] = (-1, -1)
    return distances[-1][-1], directions
```

```
def levenshtein(s, t):
    distance, directions = distance_edition_bis(s, t)
    i = len(s)
    j = len(t)
    sf = "" # s formaté et à l'envers
    tf = "" # t formaté et à l'envers
    while i != 0 or j != 0:
        if directions[i][j] == (-1, 0):
            sf += s[i-1]
            tf += "-"
            i -= 1
        elif directions[i][j] == (0, -1):
            sf += "-"
            tf += t[j-1]
            j -= 1
        else:
            sf += s[i-1]
            tf += t[j-1]
            i -= 1
            j -= 1
    return sf[::-1], tf[::-1]
```

Il s'agit exactement de l'exercice 4 du TP 4, en remplaçant les impressions à la fin par le renvoi des deux chaînes construites par la fonction `levenshtein`, sachant qu'on pose tous les coûts à un pour simplifier.

L'exercice précédent était une version simplifiée sans les directions, par ailleurs.

Exercice 3.4

```
def matrice_distances(chaines):
    n = len(chaines)
    rep = [[0 for i in range(n)] for j in range(n)]
    for i in range(n):
        for j in range(i):
            rep[i][j] = distance_edition(chaines[i], chaines[j])
            rep[j][i] = rep[i][j] # Par principe !
    return rep
```

Exercice 3.5

```
def distmin(matrice):
    rep = (0, 1)
    for i in range(len(matrice)):
        for j in range(i):
            if matrice[i][j] < matrice[rep[0]][rep[1]]:
                rep = (i, j)
    return rep
```

Exercice 3.6

Il s'agit d'un algorithme glouton.

Exercice 3.7

```
def kNN_genes(especes, sequence, k):
    distances = []
    for (seq, esp) in especes:
        distances.append((distance_edition(seq, sequence), esp))
    distances.sort()
    k_plus_proches = dict()
    for i in range(k):
        espece = distances[i][1]
        if espece not in k_plus_proches:
            k_plus_proches[espece] = 1
        else:
            k_plus_proches[espece] += 1
    maxi = espece # La variable survit à la boucle. Mais distances[0][1] fonctionne aussi.
    for cle in k_plus_proches:
        if k_plus_proches[cle] > k_plus_proches[maxi]: # Tant pis pour les égalités !
            maxi = cle
    return maxi
```

Partie 4

Exercice 4.1

```
def grille_vide()
    return [[[0 for k in range(4)] for j in range(4)] for i in range(4)]
```

Exercice 4.2

```
def piquets_possibles(grille):
    rep = []
    for i in range(16): # Première astuce pour simuler deux boucles.
        if grille[i//4][i%4][3] == 0:
            rep.append((i//4, i%4))
    return rep
```

Exercice 4.3

Version tenant compte de la suggestion, en plusieurs fonctions : la première vérifie que tous les éléments d'une liste (de taille supposée quatre vu les appels) sont égaux et non nuls, car leur somme est plus ou moins quatre ; la deuxième vérifie qu'à partir d'une coordonnée de départ et d'un vecteur de déplacement, le vecteur est non nul et qu'en le suivant trois fois on reste dans la zone autorisée de la grille ; la troisième teste toutes les coordonnées de départ et les vecteurs de déplacement.

```
def tous_egaux_non_nuls(ligne):
    if abs(sum(ligne)) == 4:
        return ligne[0]
    return 0

def test_alignement(i, j, k, di, dj, dk):
    if di == dj == dk == 0:
        return False
    return 0 <= i + 3*di <= 3 and 0 <= j + 3*dj <= 3 and 0 <= k + 3*dk <= 3
```

```
import itertools

def alignement(grille):
    for i, j, k in itertools.product(range(4), repeat=3): # trois boucles for imbriquées
        for di, dj, dk in itertools.product([-1, 0, 1], repeat=3): # idem
            if test_alignement(i, j, k, di, dj, dk):
                ligne = [grille[i+x*di][j+x*dj][k+x*dk] for x in range(4)]
                test = tous_egaux_non_nuls(ligne)
                if test != 0:
                    return test
    return 0
```

Exercice 4.4

```
def coups_gagnants(grille, joueur):
    rep = []
    for i in range(4):
        for j in range(4):
            if grille[i][j][3] == 0:
                indice = 3
                while indice > 0 and grille[i][j][indice-1] == 0:
                    indice -= 1
                grille[i][j][indice] = joueur
                test = alignement(grille)
                grille[i][j][indice] = 0 # Ne pas oublier d'annuler ou travailler sur une copie !
                if test == joueur:
                    rep.append((i, j))
    return rep
```

Exercice 4.5

```
def coups_perdants(grille, joueur):
    rep = []
    for i in range(4):
        for j in range(4):
            if grille[i][j][2] == 0:
                indice = 2
                while indice > 0 and grille[i][j][indice-1] == 0:
                    indice -= 1
                grille[i][j][indice] = joueur
                test1 = alignement(grille)
                grille[i][j][indice+1] = -joueur
                test2 = alignement(grille) # éventuellement lancé pour rien
                grille[i][j][indice] = 0
                grille[i][j][indice+1] = 0
                if test1 == 0 and test2 == -joueur:
                    rep.append((i, j))
    return rep
```

Le test ne se fait que sur des piquets où il y a encore deux places, en toute logique.

Exercice 4.6

```
def coup_gagnant_apres(grille, joueur):
    rep = []
    test0 = coups_gagnants(grille, joueur)
    if test0 != []:
        return test0[0]
    coups = piquets_possibles(grille)
    test1 = coups_gagnants(grille, -joueur)
    test1bis = coups_perdants(grille, joueur)
    if len(test1) > 1: # Impossible de tout bloquer, c'est perdu.
        return (-1, -1)
    if len(test1) == 1: # Il faut bloquer.
        coups = test1
    vrais_coups = []
    for coup in coups:
        if coup not in test1bis: # Le coup serait exclu sinon car perdant.
            vrais_coups.append(coup)
    for (i, j) in vrais_coups: # Toutes possibilités restantes, si vide on renverra forcément []
        indice = 3
        while indice > 0 and grille[i][j][indice-1] == 0:
            indice -= 1
        grille[i][j][indice] = joueur
        test2 = coups_gagnants(grille, joueur)
        test2bis = coups_perdants(grille, -joueur)
        if len(test2) > 1 or len(test2) == 1 and test2[0] in test2bis\
          or all(coup in test2bis for coup in piquets_possibles(grille)):
            grille[i][j][indice] = 0 # Annuler là aussi !
            rep.append((i, j))
    return rep
```

Exercice 4.7

(La correction est dans le cours.)

Partie 5

Exercice 5.1

Quatre pièces ayant deux côtés, les orientations étant indépendantes, cela donne seize positions.

Exercice 5.2

En considérant l'invariance par la substitution pile / face et vice-versa, la moitié des positions disparaissent, il en reste donc huit.

Exercice 5.3

Les positions dépendent alors simplement des nombres d'occurrences d'une même orientation de pièce, il y en a quatre : toutes les pièces identiques, une pièce d'une orientation et trois de l'autre, deux pièces de chaque orientation, les pièces orientées de la même façon étant adjacentes, et deux pièces de chaque orientation, les pièces orientées de la même façon étant dans des coins opposés.

Exercice 5.4

En excluant le mouvement consistant à ne rien faire, les catégories de mouvement sont proches des positions possibles : retourner une pièce, en retourner deux adjacentes ou en retourner deux dans des coins opposés.

Exercice 5.5

Position « une pièce d'une orientation » :
- Retourner une pièce → positions possibles : toutes les pièces identiques, deux de chaque adjacentes, deux de chaque opposées.
- Retourner deux pièces adjacentes → seule position possible : une pièce
- Retourner deux pièces opposées → seule position possible : une pièce

Position « deux pièces adjacentes » :
- Retourner une pièce → seule position possible : une pièce.
- Retourner deux pièces adjacentes → positions possibles : toutes les pièces identiques, deux pièces opposées
- Retourner deux pièces opposées → seule position possible : deux pièces adjacentes

Position « deux pièces opposées » :
- Retourner une pièce → seule position possible : une pièce.
- Retourner deux pièces adjacentes → seule position possible : deux pièces adjacentes
- Retourner deux pièces opposées → seule position possible : toutes les pièces identiques

Exercice 5.6

Pour faciliter la lecture, on va utiliser les notations 1 (une pièce), 2a (deux pièces adjacentes) et 2o (deux pièces opposées) pour les positions et les mouvements. Au début, positions possibles : 1, 2a, 2o.

Mouvement 1 : 2o

Positions possibles (si ce n'est pas gagné) : 1, 2a

Mouvement 2 : 2a

Positions possibles (si ce n'est pas gagné) : 1, 2o

Mouvement 3 : 2o

Seule position possible (si ce n'est pas gagné) : 1

Mouvement 4 : 1

Positions possibles (si ce n'est pas gagné) : 2a, 2o

Mouvement 5 : 2o

Seule position possible (si ce n'est pas gagné) : 2a

Mouvement 6 : 2a

Seule position possible (si ce n'est pas gagné) : 2o

Mouvement 7 : 2o

À ce stade, si ce n'était toujours pas gagné auparavant, c'est gagné.

Partie 6

Exercice 6.1

On note A, B, C, D, E, F et G les booléens.

On écrira « tester AB » pour signifier « tester si A et B sont égaux ».

Tester respectivement AB CD EF (laisser G de côté), on compte le nombre de couples de booléens différents.

0 couple : tester A et C. S'ils sont égaux, ils sont majoritaires, mais s'ils sont différents, E et F sont majoritaires.

1 couple (disons EF sans perte de généralité) : tester A et C. S'ils sont égaux, ils sont majoritaires, mais s'ils sont différents, G est majoritaire.

2 couples : le non-couple est majoritaire.

3 couples : G est majoritaire.

Exercice 6.2

Considérons toutes les stratégies possibles en trois coups, à renommage près des booléens, et construisons pour toutes les stratégies un résultat qui ne permettra pas de conclure.

La première comparaison sera forcément AB, en particulier. On considère que ce sera une égalité dans la construction de notre contre-exemple.

Une stratégie possible est de tester AC, une autre est de tester CD.

Si on teste AC, on considère que ce sera une différence. Dans ce cas-là, on a détecté un groupe de trois booléens répartis deux et un, et aucun test ne permet d'identifier par la suite un booléen majoritaire en une fois (on peut tout tester pour s'en convaincre).

Si on teste CD, on considère aussi que ce sera une égalité. Dans ce cas-là, on a détecté deux groupes de deux booléens identiques, sans savoir qu'ils sont identiques entre eux. Si la troisième comparaison est entre ces deux-là, le résultat sera une différence et ne permettra pas de conclure, si elle est entre un des quatre et un cinquième ou entre deux des trois restants, elle ne permettra pas de conclure non plus, d'où l'impossibilité de régler la question en trois comparaisons dans le pire des cas.