Correction du DS 2

Julien Reichert

Partie 1

Question 1

```
def taille_chemins(mat):
    return len(mat) + len(mat[0]) - 1
```

Question 2

```
def chemin_coherent(l):
    if l[0] != (1, 1):
        return False
    for i in range(1, len(l)):
        if l[i] not in [(l[i-1][0]+1, l[i-1][1]), (l[i-1][0], l[i-1][1]+1)]:
            return False
    return True
```

Question 3

```
def itineraire_depuis_chemin(l):
    rep = []
    x, y = 1, 1
    for i in range(1, len(l)):
        rep.append(l[i] == (x, y+1))
        x, y = l[i]
    return rep
```

Question 4

```
def chemin_depuis_itineraire(l):
    rep = [(1, 1)]
    pos = [1, 1]
    for ori in l:
        pos[ori] += 1
        rep.append((pos[0], pos[1]))
    return rep
```

Question 5

```
def entier_depuis_liste(l):
    rep = 0
    for i in range(len(l)-1, -1, -1):
        rep = 2 * rep + l[i]
    return rep
```

Question 6

```
def liste_depuis_entier(n, taille):
    rep = [False] * taille
    for i in range(taille):
        rep[i] = (n % 2 == 1)
        n //= 2
        if n == 0:
            return rep
    raise ValueError("Pas assez de place pour stocker tout n")
```

Question 7

```
def itineraire_coherent(mat, l):
    nombre_elts = [0, 0]
    for elt in l:
        nombre_elts[elt] += 1
    return nombre_elts[0] == len(mat)-1 and nombre_elts[1] == len(mat[0])-1
```

Question 8

```
def valeur_itineraire(mat, l):
    rep = mat[0][0]
    pos = [0, 0]
    for ori in l:
        pos[ori] += 1
        rep += mat[pos[0]][pos[1]]
    return rep
```

Question 9

```
def somme_maximale_bruteforce(mat):
    def somme(i, j):
        if i == len(mat)-1 and j == len(mat[0])-1:
            return mat[i][j]
        if i == len(mat)-1:
            return mat[i][j] + somme(i, j+1)
        if j == len(mat[0])-1:
            return mat[i][j] + somme(i+1, j)
        return mat[i][j] + max(somme(i+1, j), somme(i, j+1))
    return somme(0, 0)
```

Question 10

Soient n le nombre de lignes de la matrice et m son nombre de colonnes.

Tous les chemins sont testés, et leur nombre est le nombre de façons de choisir $n-1$ moments où on descend parmi $n-1+m-1$ étapes, soit $\binom{n+m-2}{n-1}$, qui est une complexité exponentielle en les dimensions de la matrice.

C'est beaucoup trop et en pratique l'utilisation d'une récursion risque fort de déclencher un dépassement de pile pour des matrices que l'on pourrait rencontrer comme arguments.

La complexité en espace est également élevée (toujours en rapport avec la pile d'appels), donc à ce stade pour éviter que Python cause un problème on peut simuler la pile soi-même en ajoutant quelques lignes, ce qui permet de se passer de la récursion.

Question 11

```
def somme_maximale_glouton1(mat):
    def somme(i, j):
        if i == len(mat)-1 and j == len(mat[0])-1:
            return mat[i][j]
        if i == len(mat)-1:
            return mat[i][j] + somme(i, j+1)
        if j == len(mat[0])-1 or mat[i+1][j] >= mat[i][j+1]:
            return mat[i][j] + somme(i+1, j)
        return mat[i][j] + somme(i, j+1)
    return somme(0, 0)
```

Question 12

Cette fois-ci un seul chemin est considéré, donc la complexité passe en $\mathcal{O}(m+n)$, toujours avec les mêmes notations.

C'est bien meilleur, mais l'optimalité fait désormais défaut, comme on va le voir tout de suite.

Question 13

Soit la matrice de deux colonnes et $n > 3$ lignes dont tous les éléments de la première colonne sont des 1 et tous les éléments de la deuxième colonne sont des 0 sauf celui à la première ligne qui est un 2. Alors l'algorithme glouton fera d'abord aller à droite, puis il faudra forcément descendre jusqu'à la fin, pour une somme de 3, alors que descendre constamment donne une somme de n.

Question 14

```
def maxmat(mat):
    rep = 0, 0 # Attention, il faudra que n'importe quelle valeur remplace ceci, et exclure l'arrivée
    for i in range(len(mat)):
        for j in range(len(mat[0])):
            if rep == (0, 0): # On ne peut pas initialiser à (0, 1) s'il n'y a qu'une colonne
                rep = i, j
            elif (i, j) != (len(mat)-1, len(mat[0])-1) and mat[i][j] > mat[rep[0]][rep[1]]:
    return rep

def somme_maximale_glouton2(mat):
    imax, jmax = maxmat(mat)
    mat1 = [[mat[i][j] for j in range(jmax+1)] for i in range(imax+1)]
    mat2 = [[mat[i][j] for j in range(jmax, len(mat[0]))] for i in range(imax, len(mat))]
    somme1 = somme_maximale_glouton1(mat1)
    somme2 = somme_maximale_glouton1(mat2)
    return somme1 + somme2 - mat[imax][jmax]
```

Question 15

On garde une fois de plus les mêmes notations.

Le calcul du maximum de la matrice est en $\mathcal{O}(mn)$. Par la suite, on construit avec la même complexité les deux sous-matrices. La consommation en mémoire n'était en pratique pas nécessaire, mais adapter la fonction `somme_maximale_glouton` aurait nécessité plus de temps et d'énergie. Les deux appels à la fonction en question sont de complexité linéaire en un peu moins que les dimensions de la matrice, ce qui donne un total restant à $\mathcal{O}(mn)$.

En bref, ce n'est pas plus cher mais plus compliqué, et on a l'impression d'optimiser un peu... mais en fait...

Question 16

Le contre-exemple précédent reste valable, car il n'y a qu'un chemin possible une fois le maximum calculé et considéré comme un point de passage nécessaire.

On pourrait par ailleurs trouver des cas où un des algorithmes gloutons donne une réponse strictement meilleure que l'autre et vice-versa.

Question 17

```
def compatibles(cpl1, cpl2):
    return cpl1[0] <= cpl2[0] and cpl1[1] <= cpl2[1] or cpl1[0] >= cpl2[0] and cpl1[1] >= cpl2[1]
```

Il s'avère qu'on va organiser les calculs de l'exercice suivant pour ne pas se servir de cette dernière fonction...

Question 18

Pour éviter de consommer de l'espace cette fois, on écrit une version de `maxmat` et de `somme_maximale_glouton1` qui travaillent dans un sous-rectangle de la matrice.

```
def maxmatbis(mat, id, jd, ia, ja):
    rep = id, jd
    for i in range(id, ia+1):
        for j in range(jd, ja+1):
            if rep == (id, jd) or (i, j) != (ia, ja) and mat[i][j] > mat[rep[0]][rep[1]]:
                rep = i, j
    return rep

def points_passage(mat, n):
    l = [(0, 0), (len(mat)-1, len(mat[0])-1)]
    possible = True
    while possible and len(l) < n+2:
        possible = False
        opt = None
        for indice in range(len(l)-1):
            id, jd = l[indice]
            ia, ja = l[indice+1]
            if id < ia and jd < ja: # Sinon c'est tout droit !
                test = maxmatbis(mat, id, jd, ia, ja)
                if opt is None or mat[test[0]][test[1]] > mat[opt[0]][opt[1]]:
                    opt = test
                    position = indice+1
        if opt is not None:
            l.insert(position, opt)
    return l

def somme_maximale_glouton1bis(mat, id, jd, ia, ja):
    def somme(i, j):
        if i == ia and j == ja:
            return mat[i][j]
        if i == ia:
            return mat[i][j] + somme(i, j+1)
        if j == ja or mat[i+1][j] >= mat[i][j+1]:
            return mat[i][j] + somme(i+1, j)
        return mat[i][j] + somme(i, j+1)
    return somme(id, jd)
```

```
def somme_maximale_glouton3(mat, n):
    l = points_passage(mat, n)
    id, jd = 0, 0
    somme = mat[0][0]
    for i in range(1, len(l)):
        ia, ja = l[i]
        somme -= mat[id][jd]
        somme += somme_maximale_glouton1bis(mat, id, jd, ia, ja)
        id, jd = ia, ja
    return somme
```

Question 19

Cette fois-ci, l'ambigüité de n force à changer de notations. On écrira t le nombre total d'éléments dans la matrice, qui était jusque là nm.

Il y a cette fois (au plus) n calculs de maximum en temps linéaire puis le même travail que précédemment, fractionné en un certain nombre d'étapes.

La fonction `points_passage` est donc en temps $\mathcal{O}(nt)$, sachant que les insertions ont une complexité $\mathcal{O}(n^2)$ non mentionnée car le nombre effectif d'insertions ne peut pas dépasser le nombre de lignes plus le nombre de colonnes, en pratique, en particulier c'est moins que t.

Le reste du travail est de complexité cumulée de l'ordre de t, en tant que somme de calculs de complexité linéaire en la taille de la zone considérée, sachant que toutes ces zones ont seulement un coin en commun, justifiant l'affirmation en début de phrase.

Pour des valeurs fixées de n, ce n'est pas si cher par rapport aux autres algorithmes gloutons, mais...

Question 20

C'est toujours le même contre-exemple, car le premier maximum calculé ruine tout espoir d'optimalité. Par la suite, il n'y a plus de choix possible donc quelle que soit la valeur de n la liste se limitera aux trois coins.

Question 21

```
def maxsomme(mat):
    nl = len(mat)
    nc = len(mat[0])
    maxsommes = [[None for _ in range(nc)] for _ in range(nl)]
    maxsommes[0][0] = mat[0][0]
    for i in range(1, nl):
        maxsommes[i][0] = maxsommes[i-1][0] + mat[i][0]
    for i in range(1, nc):
        maxsommes[0][i] = maxsommes[0][i-1] + mat[0][i]
    for i in range(1, nl):
        for j in range(1, nc):
            maxsommes[i][j] = mat[i][j] + max(maxsommes[i-1][j], maxsommes[i][j-1])
    return maxsommes[-1][-1]
```

Question 22

Chaque case n'est considérée qu'une fois avec un traitement constant, la complexité est donc linéaire en le nombre total d'éléments de la matrice. La complexité en espace est également de l'ordre de la taille de la matrice (plus quelques variables résiduelles).

Question 23

```
def maxsomme_avec_itineraire(mat):
    nl = len(mat)
    nc = len(mat[0])
    maxsommes = [[None for _ in range(nc)] for _ in range(nl)]
    directions = [[None for _ in range(nc)] for _ in range(nl)]
    maxsommes[0][0] = mat[0][0]
    for i in range(1, nl):
        maxsommes[i][0] = maxsommes[i-1][0] + mat[i][0]
        directions[i][0] = False # monter
    for i in range(1, nc):
        maxsommes[0][i] = maxsommes[0][i-1] + mat[0][i]
        directions[0][i] = True # aller à gauche
    for i in range(1, nl):
        for j in range(1, nc):
            if maxsommes[i-1][j] > maxsommes[i][j-1]:
                maxsommes[i][j] = mat[i][j] + maxsommes[i-1][j]
                directions[i][j] = False
            else:
                maxsommes[i][j] = mat[i][j] + maxsommes[i][j-1]
                directions[i][j] = True
    itineraire = []
    position = [nl-1, nc-1]
    while position != [0, 0]:
        itineraire.append(directions[position[0]][position[1]])
        if directions[position[0]][position[1]]:
            position[1] -= 1
        else:
            position[0] -= 1
    itineraire.reverse()
    return maxsommes[-1][-1], itineraire
```

Question 24

En pratique, les informations nécessaires pour calculer le contenu d'une cellule de `maxsommes` sont la case du haut et la case de droite. On peut alors calculer simplement la première ligne et la première colonne, en déduire la deuxième ligne et la deuxième colonne puis oublier la première ligne et la première colonne, etc.

L'espace nécessaire devient alors un $\mathcal{O}(n+m)$.

Ceci étant, récupérer le chemin n'est plus aussi simple car on ne sait pas parmi ce qu'on oublie ce qui fera finalement partie du chemin.

Une possibilité est de calculer la réponse en écrasant les informations non nécessaires jusqu'à atteindre la case finale, mémoriser le dernier pas, le stocker, et tout recommencer avec le reste de la mémoire réinitialisée, jusqu'à l'avant-dernière position, et ainsi de suite. La complexité en temps passerait cependant à $\mathcal{O}(n^2m^2)$. Pour éviter une explosion de la complexité en temps, il faut déjà passer par une stratégie « diviser pour régner », ce qui donne une complexité $\mathcal{O}(nm)$ en temps (preuve difficile) et en espace. L'algorithme associé est considéré comme hors de portée du tronc commun.

Partie 2

Exercice 1

On aurait pu utiliser la programmation dynamique si l'ordre avait été à prendre en compte. Ici on fait simplement deux boucles.

Le principe est de fixer le nombre d'essais transformés, puis de fixer le nombre d'essais non transformés et de voir si ce qui reste peut se limiter à des drops (ou pénalités).

```
def nb_facons_rugby(n):
    rep = 0
    for n1 in range(n, -1, -7): # en enlevant les essais transformés
        for n2 in range(n1, -1, -5): # en enlevant les essais non transformés
            if n2 % 3 == 0: # On peut faire un certain nombre de drops
                rep += 1
    return rep
```

Exercice 2

```
def equipe_depuis_joueur(joueur):
    for equipe in joueurs:
        if joueur in joueurs[equipe]:
            return equipe
    raise ValueError("Ceci n'aurait pas dû se produire !")
```

La complexité est linéaire en le nombre total de joueurs, car il s'agit de parcourir des listes. Il aurait pu être intéressant d'utiliser une structure d'ensemble / de dictionnaire pour les valeurs associées à chaque équipe, pour accélérer virtuellement la recherche.

Exercice 3

```
def dictionnaire_equipes():
    rep = dict()
    for equipe in joueurs:
        for joueur in joueurs[equipe]:
            rep[joueur] = equipe
    return rep
```

On considère par commodité que désormais une variable globale est créée par l'instruction suivante :

```
equipes = dictionnaire_equipes()
```

Exercice 4

```
def coherence(cr):
    for i in range(len(cr)):
        if cr[i][1] == "transformation":
            if i == 0 or equipes[cr[i][0]] != equipes[cr[i-1][0]] or cr[i-1][1] != "essai":
                return False
    return True
```

Exercice 5

On ne sait pas a priori quelles équipes sont concernées avec l'argument fourni, mais il faudrait en tout cas qu'il y en ait au plus deux (s'il n'y en a qu'une, c'est que l'Italie joue...).

```
def coherencebis(cr):
    eq = dict()
    for j, _, _ in cr:
        eq[equipes[j]] = True
    return len(eq) <= 2
```

Exercice 6

Une autre variable globale utile pour la suite :

```
points_par_action = { "essai" : 5, "transformation" : 2, "drop" : 3, "penalite" : 3 }
# on évite les accents
```

Pour l'exercice, on passera par un dictionnaire, c'est plus pratique. Ensuite, pour la conformité avec l'énoncé, on construira le couple associé.

```
def score_match(cr, e1, e2):
    resultat = { e1 : 0, e2 : 0 }
    for j, t, _ in cr:
        resultat[equipes[j]] += points_par_action[t]
    return resultat[e1], resultat[e2]
```

Exercice 7

```
def points_match(cr, e1, e2):
    points = { e1 : 0, e2 : 0 }
    essais = { e1 : 0, e2 : 0 }
    resultat = { e1 : 0, e2 : 0 }
    for j, t, _ in cr:
        resultat[equipes[j]] += points_par_action[t]
        if t == "essai":
            essais[equipes[j]] += 1
    if resultat[e1] > resultat[e2]:
        points[e1] = 4
        if resultat[e1] <= resultat[e2] + 7:
            points[e2] = 1
    elif resultat[e2] < resultat[e2]:
        points[e2] = 4
        if resultat[e2] <= resultat[e1] + 7:
            points[e1] = 1
    else:
        points[e1] = 2
        points[e2] = 2
    if essais[e1] >= 4:
        points[e1] += 1
    if essais[e2] >= 4:
        points[e2] += 1
    return points[e1], points[e2]
```

Exercice 8

```
SELECT equipe_nom
FROM Equipe
```

Exercice 9

```
SELECT joueur_nom
FROM Equipe JOIN Joueur ON Equipe.equipe_id = Joueur.equipe_id
WHERE equipe_nom = "France"
```

Exercice 10

```
SELECT DISTINCT joueur_nom
FROM Equipe JOIN Joueur ON Equipe.equipe_id = Joueur.equipe_id
            JOIN Deroulement ON Joueur.joueur_id = Deroulement.joueur_id
WHERE equipe_nom = "France" AND match_evenement = "essai"
```

Exercice 11

```
SELECT MIN(match_moment), MAX(match_moment)
FROM Deroulement
```

Exercice 12

```
SELECT equipe_nom, SUM(points_points) AS score
FROM Equipe JOIN Joueur ON Equipe.equipe_id = Joueur.equipe_id
            JOIN Deroulement ON Joueur.joueur_id = Deroulement.joueur_id
            JOIN Points ON Deroulement.match_evenement = Points.match_evenement
WHERE match_id =
(
    SELECT match_id
    FROM Match JOIN Equipe AS E1 ON match_equipe1 = E1.equipe_id
               JOIN Equipe AS E2 ON match_equipe2 = E2.equipe_id
    WHERE E1.equipe_nom = "France" AND E2.equipe_nom = "Angleterre"
          OR E2.equipe_nom = "France" AND E1.equipe_nom = "Angleterre"
)
GROUP BY equipe_nom
```